MANNING

全球有超过 100 000 名开发者使用本书来学习 Spring
畅销经典 Spring 技术图书，针对 Spring 5 全面升级

异步图书
www.epubit.com

Spring实战

（第5版）

Spring IN ACTION

5TH EDITION



【美】克雷格·沃斯
（Craig Walls）著
张卫滨 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：Spring实战（第5版）

ISBN：978-7-115-52792-9

本书由人民邮电出版社发行数字版。

版　权

著　　　　［美］克雷格•沃斯（Craig Walls）

译　　　　张卫滨

责任编辑　陈冀康

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

本书是一本经典而实用的畅销Spring学习指南。

第5版涵盖了Spring 5.0和Spring Boot 2.0里程碑式的更新。全书分为5个部分，共19章。第1部分（第1～5章）涵盖了构建Spring应用的基础话题。第2部分（第6～9章）讨论如何将Spring应用与其他应用进行集成。第3部分（第10～12章）探讨Spring对反应式编程提供的全新支持。第4部分（第13～15章）拆分单体应用模型，介绍Spring Cloud和微服务开发。第5部分（第16～19章）讨论如何为应用投入生产环境做准备以及如何进行部署。

本书既适合刚开始学习Spring Boot和Spring框架的Java开发人员快速上手，也适合经验丰富的Spring开发人员学习Spring的新特性，尤其适用于企业级Java开发人员。

译者序

不知不觉间，已经参与了3个版本的《Spring实战》翻译。2007年春天，当时J2EE Without EJB的风潮刚刚兴起，还在读研的我在天津大学图书馆借到第 1 版的《Spring实战》，当时忙于毕业，没有把这本书完整地读完，但是依赖注入、面向切面编程等理念还是深深印在了脑海中，书中经典有趣的圆桌骑士样例更是驱使我读了很多相关的历史背景材料。

屈指算来，已经过去了10多年，Spring早已经成为企业级Java开发的事实标准，Spring Boot和Spring Cloud相关的技术引领潮流，更是成为Java工程师的必备技能。《Spring实战》的作者Craig Walls不断推陈出新，将这本经典图书更新到第5版。只是，10多年以前，我恐怕做梦也想不到会与这本书有这么深的缘分。

Spring之所以能够在技术不断更新换代的IT领域长盛不衰，并且引领技术架构发展的潮流，我想这是因为它一直没有偏离Rod Johnson最初的目标。那就是，根据技术的发展，不断优化和革新，让Java应用的开发更加便利和高效。从XML配置、注解配置，再到Spring Boot的自动化配置，Spring在不断简化，开发人员需要做的额外工作越来越少。虽然Rod Johnson早已离开Spring去开创新的事业了，但是我相信Spring的这种基因还是一直在的。在可以预见的未来，Spring及其家族产品依然是值得花时间投资学习的技术。

有时候，我也会思考，真正的技术到底是什么，是某一项生僻的配置还是某个新的API？我想，这都是技术，却不是最关键的。因为这些东西都是不稳定的、易变的，想要在新知识层出不穷的领域中不被淘汰，我们更应该去追求一些内在稳定不变的知识，比如技术规范、设计原理等。所以，希望本书的读者能够通过这本入门的读物，去更多地探究一些Spring底层的设计和实现原理。

本书第10章关于反应式编程的初稿由何品翻译，我负责统稿修改；另外，对于反应式编程的术语和规范，何品和他的团队都做了很多的工作，在此向他表示感谢。

再次感谢我的爱人和儿子，又容忍我把这几个月的业余时间都耗在了笔记本电脑的前面。

希望这本书对读者有所帮助，如果读者在阅读中遇到问题，可以通过levinzhang1981 @126.com或者微信levinzhang1981与我联系，祝阅读愉快。

张卫滨

2019年7月29日于大连

关于本书

编写《Spring实战（第5版）》的目的是让读者学会使用Spring框架、Spring Boot以及Spring生态系统中各种辅助部分构建令人赞叹的应用程序。本书首先介绍如何使用Spring和Spring Boot开发基于Web、以数据库作为后端的Java应用；随后进行必要的扩展，展现如何与其他应用进行集成、使用反应式类型进行编程，以及将应用拆分为离散的微服务；最后讨论如何准备应用的部署。

尽管Spring生态系统中的每个项目都提供了完善的文档，但是本书所做的是所有参考文档都无法做到的事情：提供一个实用的、项目驱动的指南，将Spring的各种元素组合起来形成一个真正的应用。

谁应该阅读这本书

《Spring实战（第5版）》适用于刚刚开始学习Spring Boot和Spring框架的Java开发人员，也适用于想要超越基础知识并学习Spring新特性的经验丰富的Spring开发者。

这本书是如何组织的：路线图

本书分成了5个部分，共计19章。

第1部分涵盖构建Spring应用的基础话题。

- 第1章介绍Spring和Spring Boot以及如何初始化Spring项目。在本章中，我们迈出构建Spring应用的第一步，在本书后续各章中，我们会对这个应用进行扩展。
- 第2章讨论如何使用Spring MVC构建应用的Web层。在本章中，我们将会构建处理Web请求的控制器以及在浏览器中渲染信息的视图。
- 第3章会深入探讨Spring应用的后端，在这里数据会持久化到关系型数据库中。
- 在第4章中，我们会使用Spring Security认证用户并防止未认证的用户访问应用。
- 第5章介绍如何使用Spring Boot的配置属性功能来配置Spring应用。我们还会学习如何使用profile选择性地应用配置。

第2部分讨论如何将Spring应用与其他应用进行集成。

- 第6章延续第2章对Spring MVC的讨论，我们将会学习如何在Spring中编写REST API。
- 第7章讨论了和第6章相对立的主题，展现Spring应用如何消费REST API。
- 第8章会讨论如何使用异步通信技术让Spring应用发送和接收消息，这里会用到Java Message Service、RabbitMQ或Kafka。
- 第9章讨论如何使用Spring Integration进行声明式的应用集成。

第3部分探讨Spring对反应式编程提供的全新支持。

- 第10章介绍Reactor项目。这是一个反应式编程库，支撑了Spring 5的反应式特性。

- 第11章重新探讨REST API开发，介绍全新的Web框架Spring WebFlux。该框架借用了很多Spring MVC的理念，但是为Web开发提供了新的反应式模型。
- 第12章将会看一下如何使用Spring Data编写反应式数据持久化，我们将会读取和写入Cassandra与Mongo数据库。

第4部分将会拆分单体应用模型，介绍Spring Cloud和微服务开发。

- 第13章会深入介绍服务发现，组合使用Spring和Netflix的注册中心实现Spring微服务的注册和发现。
- 第14章将展现如何在配置服务器中实现中心化的应用配置，从而实现跨微服务共享配置。
- 第15章会介绍Hystrix的断路器模式。它能够让微服务在面临失败时更有弹性。

在第5部分中，我们将会讨论如何做好将应用投入生产环境的准备，并看一下如何进行部署。

- 第16章会介绍Spring Boot Actuator。它是Spring Boot的一个扩展，通过REST端点的形式暴露Spring应用内部的运行状况。
- 第17章将会介绍如何使用Spring Boot Admin。它是构建在Actuator之上的一个用户友好的基于浏览器的管理应用。
- 第18章将会讨论如何将Spring bean暴露为JMX MBean以及如何消费它们。
- 在第19章中，我们会看到如何将Spring应用部署到各种生产环境中。

通常来讲，刚刚接触Spring的开发人员应该从第1章开始，并按顺序阅读每一章；经验丰富的Spring开发人员可能更愿意在任何感兴趣的时

候参与进来。即便如此，每一章都是建立在前一章内容的基础上的，所以如果从中间开始阅读，那么可能会漏掉一些上下文信息。

关于代码

本书包含许多源代码的样例，有的是编号的程序清单，有的是普通文本内嵌的源码。在这两种情况下，源代码都使用固定宽度的字体排版，以便将其与普通文本分开。有时代码也会使用粗体显示，以便于强调此处代码与本章前面步骤的变更，比如为已有的代码行添加新的特性。

在许多情况下，原始源代码会重新格式化；我们添加了换行符和重新缩进，以适应书中可用的页面空间。在极少数情况下，这样做依然是不够的，在这种情况下程序清单会包括换行符(➥)。此外，当在文中描述代码的时候，源码中的注释通常会被移除。许多程序清单会有代码标注，用来突出强调重要的概念。

本书中样例的源码可以通过异步社区的本书页面下载。

其他在线资源

还需要其他帮助吗？

- Spring的Web站点有很多有用的起步指南（其中一部分就是由本书的作者编写的）。
- StackOverflow上的Spring标签和Spring Boot是一个询问Spring问题

和帮助别人的好地方。帮助解决别人的Spring问题是学习Spring的好办法。

关于作者

克雷格·沃斯（Craig Walls）是Pivotal的首席工程师。他是Spring框架的热心推动者，经常在本地用户组和会议上发言，撰写关于Spring的文章。在不琢磨代码的时候，Craig正在计划去迪士尼世界或迪士尼乐园的下一次旅行，他希望尽可能多地陪伴他的妻子和两个女儿。

关于封面

《Spring实战（第5版）》的封面人物是“Le Caraco”，也就是约旦西南部卡拉克（Karak）省的居民。该省的首府是Al-Karak，那里的山顶有座城堡，对死海和周边的平原有着极佳的视野。这幅图出自1796年出版的法国旅游图书，*Encyclopédiedes Voyages*，由J.G.St.Sauveur编写。在那时，为了娱乐而去旅游还是相对新鲜的做法，而像这样的旅游指南是很流行的，它能够让旅行家和足不出户的人们了解法国其他地区和国外的居民。

*Encyclopédiedes Voyages*中多种多样的图画生动描绘了200年前世界上各个城镇和地区的独特魅力。在那时，相隔几十千米的两个地区着装就不相同，可以通过着装判断人们究竟属于哪个地区。这本旅行指南展现了那个时代和其他历史时代的隔离感和距离感，这与我们这个运动过度的时代是截然不同的。

从那以后，服装风格发生了改变，富有地方特色的多样性开始淡化。现在，有时很难说一个洲的居民和其他洲的居民有什么不同。可能，从积极的方面来看，我们用原来文化和视觉上的多样性换来了个人风格的多变性，或者可以说是更为多样化和有趣的知识科技生活。这本旅行指南中的图片反映了两个世纪前各个地区生活的多样性，我们现在用图书封面的方式对其进行了再现。Manning出版社的员工都认为这是计算机行业中一个很有意思的创意。

前　　言

在使用了Spring 15年并编写了这本书的5个版本（暂时不算《Spring Boot实战》了）之后，你可能会认为，在为这本书撰写前言时，我很难想出一些关于Spring令人兴奋的新内容，但事实远非如此!

在Spring生态系统中，Spring、Spring Boot和所有其他项目的每个版本都发布了令人兴奋的新功能，重新点燃了开发应用程序的乐趣。Spring 5.0和Spring Boot 2.0的发布达到了一个重要的里程碑。Spring有了更多的乐趣，所以编写新版《Spring实战》是很容易的。

Spring 5的主要功能是对反应式编程的支持，包括Spring WebFlux。这是一个全新的反应式Web框架，借鉴了Spring MVC的编程模型，允许开发人员创建伸缩性更好且耗用更少线程的Web应用程序。至于Spring应用的后端，最新版本的Spring Data支持创建反应式、非阻塞的数据repository。所有这些都构建在Reactor项目之上，Reactor是一个用于处理反应式类型的Java库。

除了Spring 5新的反应式编程特性之外，Spring Boot 2提供了比以前更多的自动配置支持，以及一个完全重新设计的Actuator，用于探查和操作正在运行的应用。

更重要的是，当开发人员希望将单体应用拆分为分散的微服务时，Spring Cloud提供了一些工具，使配置和发现微服务变得容易，并增强

了微服务的功能，使它们更能抵御失败。

我很高兴地说，《Spring实战（第5版）》涵盖了所有的这些功能，甚至更多!如果你是经验丰富的老手，《Spring实战（第5版）》可以作为指南，指导你去学习Spring提供的新功能；如果你是Spring新手，那么现在是行动起来的最佳时机，本书的前几章会让你快速上手!

与Spring合作的15年是令人兴奋的。现在我已经写了5个版本的《Spring实战》，我很想和你们分享这份兴奋!

致　谢

Spring和Spring Boot所做的最令人惊奇的事情之一就是自动为应用程序提供所有的基础功能，让开发人员专注于应用程序特有的逻辑。不幸的是，对于写书这件事来说，并没有这样的魔法。是这样的吗？

在Manning，有很多人在施展“魔法”，确保这本书是最好的。特别要感谢我的项目编辑Jenny Stout以及制作团队，其中包括项目主管Janet Vail、文字编辑Andy Carroll和Frances Buran，以及校对Katie Tennant和Melody Dolab。同时也要感谢技术校对Joshua White，他的工作很全面，很有帮助。

在此过程中，我们得到了几位同行评论的反馈，他们确保了这本书没有偏离目标，涵盖了正确的内容。为此，我要感谢Andrea Barisone、Arnaldo Ayala、Bill Fly、Colin Joyce、Daniel Vaughan、David Witherspoon、Eddu Melendez、Iain Campbell、Jettro Coenradie、John Gunvaldson、Markus Matzker、Nick Rakochy、Nusry Firdousi、Piotr Kafel、Raphael Villela、Riccardo Noviello、Sergio Fernandez Gonzalez、Sergiy Pylypets、Thiago Presa、Thorsten Weber、Waldemar Modzelewski、Yagiz Erkan和Željko Trogrlić。

和往常一样，如果不是Spring工程团队成员所做的出色工作，写这本书是没有任何意义的。我惊叹于你们所创造的成果，并期待未来继续

改变软件开发的方式。

非常感谢我的同行们在No Fluff/Just Stuff巡回演讲上的发言。我从你们每个人身上学到很多。我特别要感谢Brian Sletten、Nate Schutta和Ken Kousen关于Spring的对话和邮件，这些内容塑造了这本书。

我要再次感谢Phoenicians，你们太棒了[1]。

最后，我要感谢我美丽的妻子Raymie。她是我生命中的挚爱，是我最甜蜜的梦想，也是我的灵感来源：谢谢你的鼓励，也谢谢你为这本新书做的努力。致我可爱的女儿Maisy和Madi：我为你们感到骄傲，为你们即将成为了不起的年轻女士感到骄傲。我对你们的爱超出了你们的想象，也超出了我语言所能表达的程度。

---

[1]　Phoenicians指的是远古时代的腓尼基人，他们被认为是字母系统的创建者，基于字母的所有现代语言都是由此衍生而来的。在迪士尼世界的Epcot，有名为Spaceship Earth的时光穿梭体验，我们可以了解到人类交流的历史，甚至能够回到腓尼基人的时代，在这段旅程的旁白中这样说道：如果你觉得学习字母语言很容易，就感谢腓尼基人吧，是他们发明了它。这是作者的一种幽默说法。——译者注

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供如下资源：

- 本书源代码；
- 书中彩图文件。

要获得以上配套资源，请在异步社区本书页面中点击配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

如果您是教师，希望获得教学配套资源，请在社区本书页面中直接联系本书的责任编辑。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。

详细信息　写书评　提交勘误

页码：　页内位置（行数）：　勘误印次：

字数统计

提交

与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问www.epubit.com/ selfpublish/submission即可）。

如果学校、培训机构或企业想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。



异步社区



微信服务号

第1部分　Spring基础

本书的第1部分将会介绍如何开始编写Spring应用，并在这个过程中学习Spring的基础知识。

在第 1 章中，我将简要介绍Spring和Spring Boot的核心知识，并展示在构建第一个Spring应用Taco Cloud的过程中如何初始化Spring项目。在第2章中，我们将深入研究Spring MVC，了解如何在浏览器中显示模型数据，以及如何处理和验证表单输入。我们还会介绍选择视图模板库的技巧。在第3章中，我们将向Taco Cloud应用程序添加数据持久化功能。到时候，我们将介绍如何使用Spring的JDBC模板来插入数据，以及如何使用Spring Data声明JPA repository。第4章将介绍Spring应用程序的安全性，包括自动配置Spring安全性、声明自定义用户存储、自定义登录页面以及防止跨站请求伪造（CSRF）攻击。作为第1部分的结尾，我们将在第5章中学习配置属性。我们将了解如何细粒度调整自动配置bean、让应用组件使用配置属性，以及如何使用Spring profile。

第1章　Spring起步

本章内容：

- Spring和Spring Boot的必备知识
- 初始化Spring项目
- Spring生态系统概览

尽管希腊哲学家赫拉克利特（Heraclitus）并不作为一名软件开发人员而闻名，但他似乎深谙此道。他的一句话经常被引用："唯一不变的就是变化"，这句话抓住了软件开发的真谛。

我们现在开发应用的方式和1年前、5年前、10年前都是不同的，更别提15年前了，当时Rod Johnson的图书*Expert One-on-One J2EE Design and Development*介绍了Spring框架的初始形态。

当时，最常见的应用形式是基于浏览器的Web应用，后端由关系型数据库作为支撑。尽管这种形式的开发依然有它的价值，Spring也为这种应用提供了良好的支持，但是我们现在感兴趣的还包括如何开发面向

云的由微服务组成的应用，这些应用会将数据保存到各种类型的数据库中。另外一个崭新的关注点是反应式编程，它致力于通过非阻塞操作提供更好的扩展性并提升性能。

随着软件开发的发展，Spring框架也在不断变化，以解决现代应用开发中的问题，其中就包括微服务和反应式编程。Spring还通过引入Spring Boot简化自己的开发模型。

不管你是开发以数据库作为支撑的简单Web应用，还是围绕微服务构建一个现代应用，Spring框架都能帮助你达成目标。本章是使用Spring进行现代应用开发的第一步。

1.1　什么是Spring

我知道你现在可能迫不及待地想要开始编写Spring应用了，我可以向你保证，在本章结束之前，你肯定能够开发一个简单的Spring应用。首先，我将使用Spring的一些基础概念为你搭建一个舞台，帮助你理解Spring是如何运行起来的。

任何实际的应用程序都是由很多组件组成的，每个组件负责整个应用功能的一部分，这些组件需要与其他的应用元素进行协调以完成自己的任务。当应用程序运行时，需要以某种方式创建并引入这些组件。

Spring的核心是提供了一个容器（container），通常称为Spring应用上下文（Spring application context），它们会创建和管理应用组件。这些组件也可以称为bean，会在Spring应用上下文中装配在一起，从而形

成一个完整的应用程序。这就像砖块、砂浆、木材、管道和电线组合在一起，形成一栋房子似的。

将bean装配在一起的行为是通过一种基于依赖注入（dependency injection，DI）的模式实现的。此时，组件不会再去创建它所依赖的组件并管理它们的生命周期，使用依赖注入的应用依赖于单独的实体（容器）来创建和维护所有的组件，并将其注入到需要它们的bean中。通常，这是通过构造器参数和属性访问方法来实现的。

举例来说，假设在应用的众多组件中，有两个是我们需要处理的：库存服务（用来获取库存水平）和商品服务（用来提供基本的商品信息）。商品服务需要依赖于库存服务，这样它才能提供商品的完整信息。图1.1阐述这些bean和Spring应用上下文之间的关系。

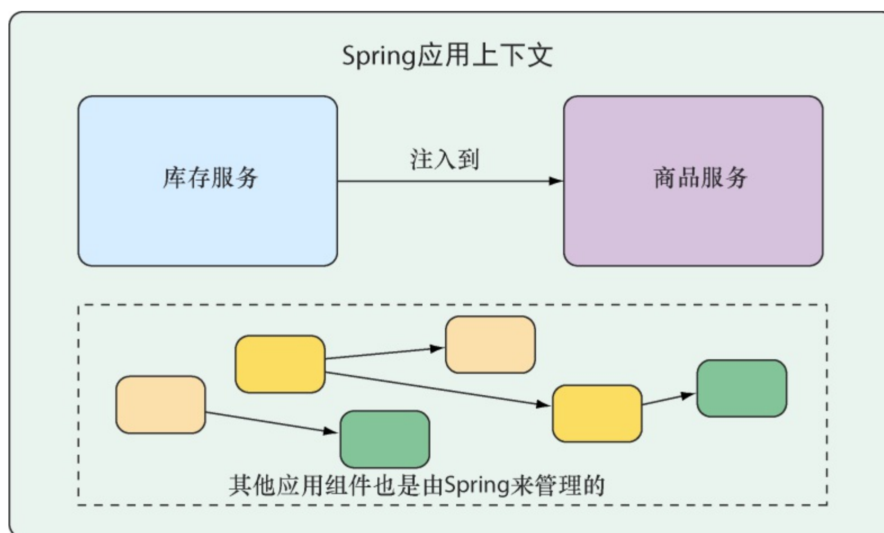


图1.1　应用组件通过Spring的应用上下文来进行管理并实现互相注入

在核心容器之上，Spring及其一系列的相关库提供了Web框架、各种持久化可选方案、安全框架、与其他系统集成、运行时监控、微服务

支持、反应式编程以及众多现代应用开发所需的特性。

在历史上，指导Spring应用上下文将bean装配在一起的方式是使用一个或多个XML文件（描述各个组件以及它们与其他组件的关联关系）。例如，如下的XML描述了两个bean，也就是InventoryService bean和ProductService bean，并且通过构造器参数将InventoryService装配到了ProductService中：

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
  <constructor-arg ref="inventoryService" />
</bean>
```

但是，在最近的Spring版本中，基于Java的配置更为常见。如下基于Java的配置类是与XML配置等价的：

```
@Configuration
public class ServiceConfiguration {
  @Bean
  public InventoryService inventoryService() {
    return new InventoryService();
  }

  @Bean
  public ProductService productService() {
    return new ProductService(inventoryService());
  }
}
```

@Configuration注解会告知Spring这是一个配置类，会为Spring应用上下文提供bean。这个配置类的方法使用@Bean注解进行了标注，表明这些方法所返回的对象会以bean的形式添加到Spring的应用上下文中

（默认情况下，这些bean所对应的bean ID与定义它们的方法名称是相同的）。

相对于基于XML的配置方式，基于Java的配置会带来多项额外的收益，包括更强的类型安全性以及更好的重构能力。即便如此，不管是使用Java还是使用XML的显式配置，只有当Spring不能进行自动配置的时候才是必要的。

在Spring技术中，自动配置起源于所谓的自动装配（autowiring）和组件扫描（component scanning）。借助组件扫描技术，Spring能够自动发现应用类路径下的组件，并将它们创建成Spring应用上下文中的bean。借助自动装配技术，Spring能够自动为组件注入它们所依赖的其他bean。

最近，随着Spring Boot的引入，自动配置的能力已经远远超出了组件扫描和自动装配。Spring Boot是Spring框架的扩展，提供了很多增强生产效率的方法。最为大家所熟知的增强方法就是自动配置（autoconfiguration），Spring Boot能够基于类路径中的条目、环境变量和其他因素合理猜测需要配置的组件并将它们装配在一起。

我非常愿意为你展现一些关于自动配置的示例代码，但是我做不到。自动配置就像风一样，你可以看到它的效果，但是我找不到代码指给你说，“看！这就是自动配置的样例！”事情发生了，组件启用了，功能也提供了，但是不用编写任何代码。没有代码就是自动装配的本质，也是它如此美妙的原因所在。

Spring Boot大幅度减少了构建应用所需的显式配置的数量（不管是XML配置还是Java配置）。实际上，当完成本章的样例时，我们会有一个可运行的Spring应用，该应用只有一行Spring配置代码。

Spring Boot极大地改善了Spring的开发，因此很难想象在没有它的情况下如何开发Spring应用。因此，本书会将Spring和Spring Boot当成一回事。我们会尽可能多地使用Spring Boot，只有在必要的时候才使用显式配置。因为Spring XML配置是一种过时的方式，所以我们主要关注Spring基于Java的配置。

闲言少叙，既然本书的名称中包含“实战”这个词，那么就开始动手吧！下面我们将会编写使用Spring的第一个应用。

1.2　初始化Spring应用

在本书中，我们将会创建一个名为Taco Cloud的在线应用，它能够订购人类所发明的一种美味，也就是墨西哥煎玉米卷（taco）[1]。当然，在这个过程中，为了达成我们的目标，我们将会用到Spring、Spring Boot以及各种相关的库和框架。

我们有多种初始化Spring应用的可选方案。尽管我可以教你手动创建项目目录结构和定义构建规范的各个步骤，但这无疑是浪费时间，我们最好将时间花在编写应用代码上。因此，我们将会学习如何使用Spring Initializr初始化应用。

Spring Initializr是一个基于浏览器的Web应用，同时也是一个REST

API，能够生成一个Spring项目结构的骨架，我们还可以使用各种想要的功能来填充它。使用Spring Initializr的几种方式如下：

- 通过地址为https://start.spring.io/的Web应用；
- 在命令行中使用curl命令；
- 在命令行中使用Spring Boot命令行接口；
- 在Spring Tool Suite中创建新项目；
- 在IntelliJ IDEA中创建新项目；
- 在NetBeans中创建新项目。

我将这些细节放到了附录中，这样就不用在这里花费很多页的篇幅介绍每种方案了。在本章和本书中，我都会向你展示如何使用我最钟爱的方式创建新项目：在Spring Tool Suite中使用Spring Initializr。

顾名思义，Spring Tool Suite是一个非常棒的Spring开发环境。它同时还提供了便利的Spring Boot Dashboard特性，这个特性是其他IDE都不具备的（至少在我编写本书的时候如此）。

如果你不是Spring Tool Suite用户，那也没有关系，我们依然可以做朋友。你可以跳转到附录中，查看最适合你的Initializr方案，以此来替换后面小节中的内容。但是，在本书中，我偶尔会提到Spring Tool Suite特有的特性，比如Spring Boot Dashboard。如果你不使用Spring Tool Suite，那么需要调整这些指令以适配你的IDE。

1.2.1　使用Spring Tool Suite初始化Spring项目

要在Spring Tool Suite中初始化一个新的Spring项目，我们首先要点击File菜单，选择New，接下来选择Spring Starter Project。图1.2展现了要查找的菜单结构。

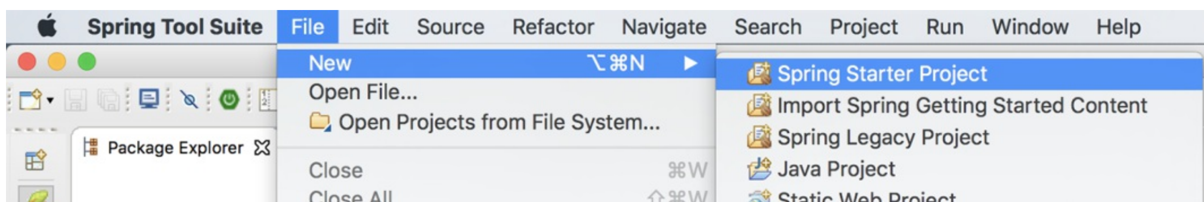


图1.2　在Spring Tool Suite中使用Initializr初始化一个新项目

在选择Spring Starter Project之后，将会出现一个新的向导对话框（见图1.3）。向导的第一页会询问一些项目的通用信息，比如项目名称、描述和其他必要的信息。如果你熟悉Maven pom.xml文件的内容，就可以识别出大多数的输入域条目最终都会成为Maven的构建规范。对于Taco Cloud应用来说，我们可以按照图1.3的样子来填充对话框。

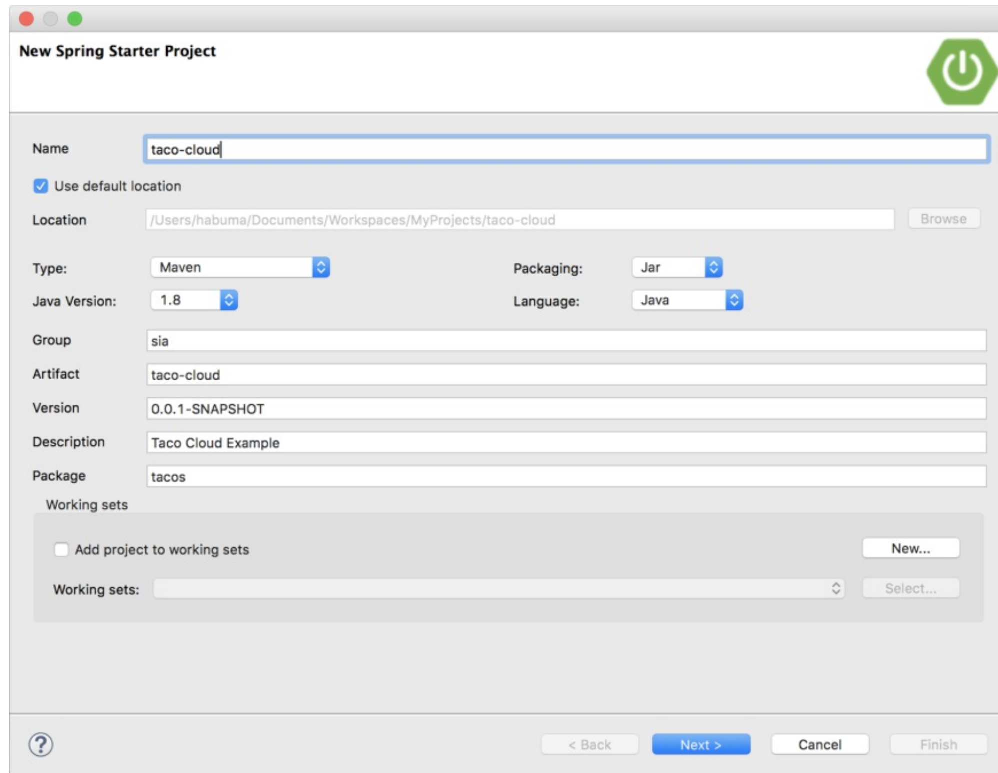


图1.3　为Taco Cloud应用指定通用的项目信息

向导的下一页会让我们选择要添加到项目中的依赖（见图1.4）。注意，在对话框的顶部，我们可以选择项目要基于哪个Spring Boot版本。它的默认值是最新的可用版本。一般情况下，最好使用这个默认的值，除非你需要使用不同的版本。

至于依赖项本身，你可以打开各个区域并查找所需的依赖项，也可以在Available顶部的搜索框中对依赖进行搜索。对于Taco Cloud应用来说，我们最初的依赖项如图1.4所示。

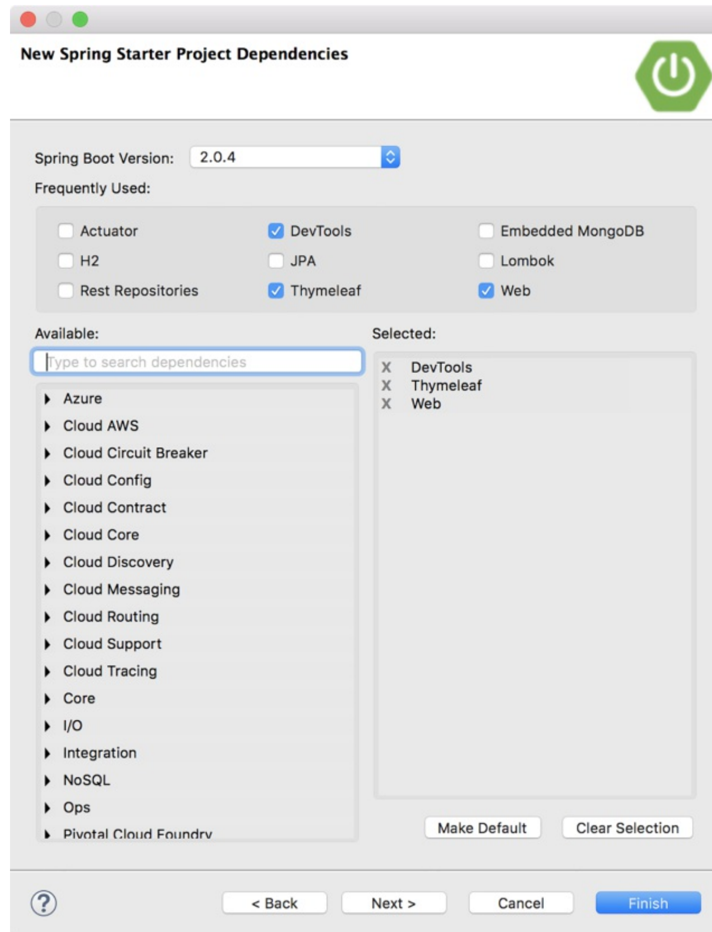


图1.4　选择Starter依赖

现在，你可以点击Finish来生成项目并将其添加到工作空间中。但是，如果你还想多体验一些，那么可以再次点击Next，看一下新Starter项目向导的最后一页，如图1.5所示。

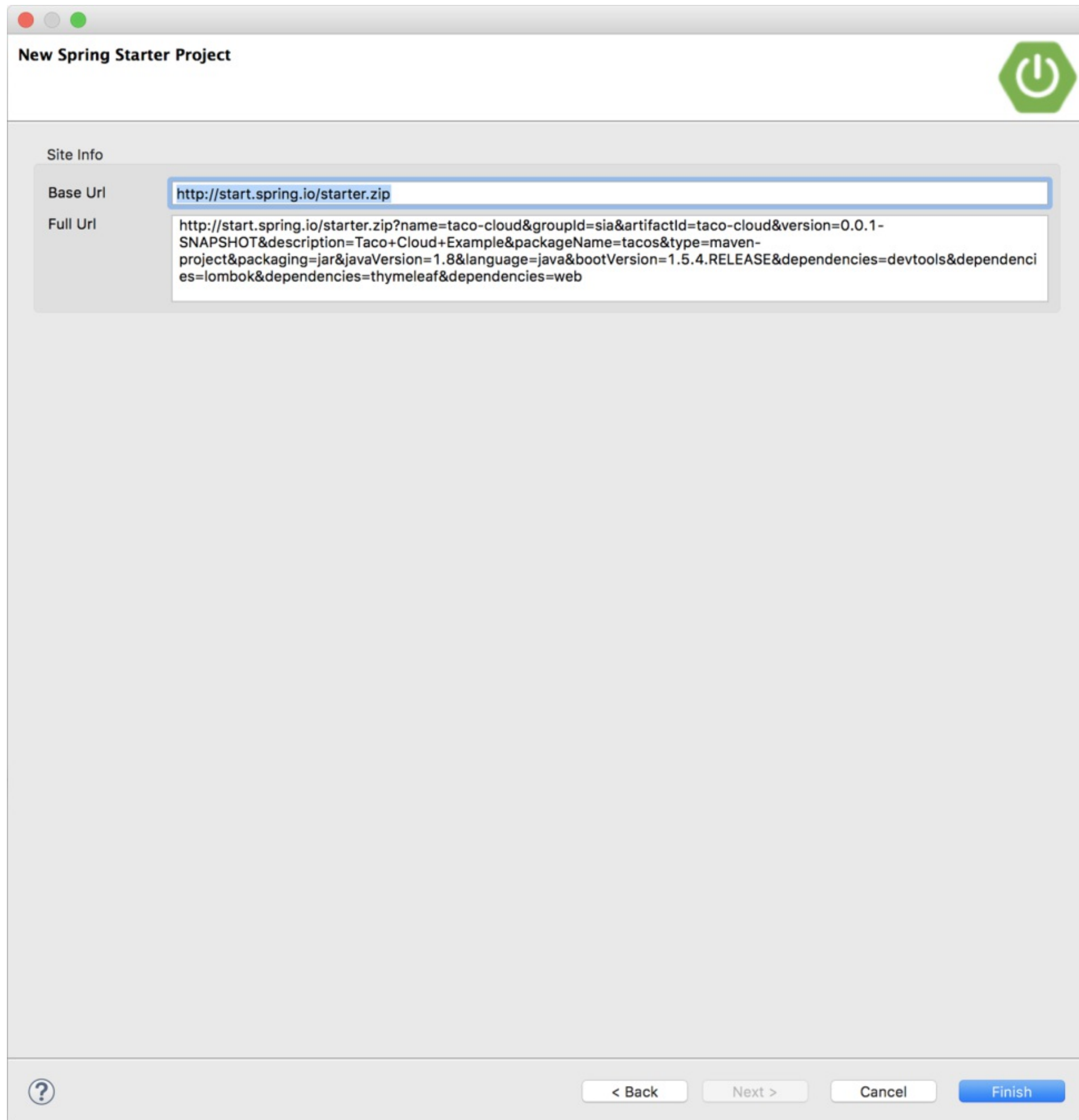


图1.5　指定备用的Initializr地址

默认情况下，新项目的向导会调用Spring Initializr来生成项目。通常情况下，没有必要覆盖默认值，这也是我们可以在向导的第二页直接点击Finish的原因。但是，如果你基于某种原因托管了自己的Initializr克隆版本（可能是本地机器上的副本或者公司防火墙内部运行的自定义克隆版本），那么你可能需要在点击Finish之前修改Base Url输入域，使其指向自己的Initializr实例。

在点击Finish之后，项目会从Initializr下载并加载到工作空间中。此时，要等待它加载和构建，然后你就可以开始开发应用功能了。下面我们看一下Initializr都为我们提供了什么。

1.2.2　检查Spring项目的结构

项目加载到IDE中之后，我们将其展开，看一下其中都包含什么内容。图1.6展现了Spring Tool Suite中已展开的Taco Cloud项目。

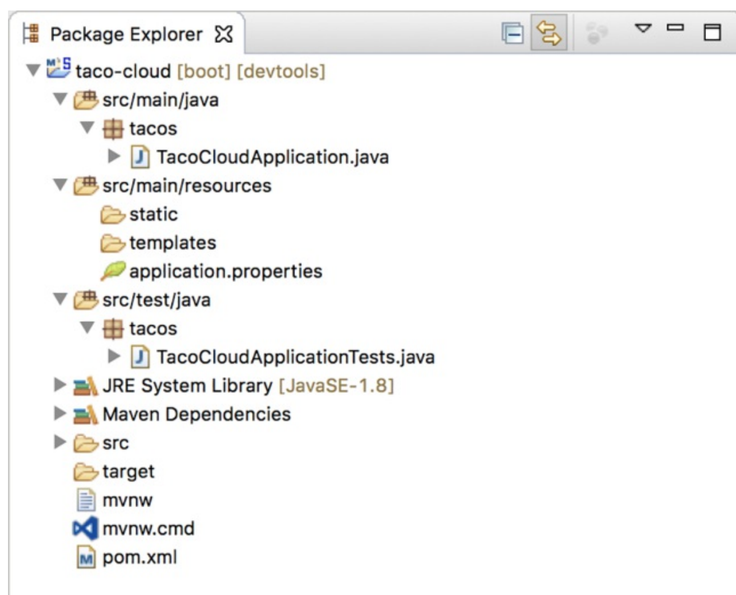


图1.6　Spring Tool Suite中所展现的初始Spring项目结构

你可能已经看出来了，这就是一个典型的Maven或Gradle项目结构，其中应用的源码放到了“src/main/java”中，测试代码放到了“src/test/java”中，而非Java的资源放到了“src/main/resources”。在这个项目结构中，我们需要注意以下几点。

- mvnw和mvnw.cmd：这是Maven包装器（wrapper）脚本。借助这些

脚本，即便你的机器上没有安装Maven，也可以构建项目。

- pom.xml：这是Maven构建规范，随后我们将会深入介绍该文件。
- TacoCloudApplication.java：这是Spring Boot主类，它会启动该项目。随后，我们会详细介绍这个类。
- application.properties：这个文件起初是空的，但是它为我们提供了指定配置属性的地方。在本章中，我们会稍微修改一下这个文件，但是我会将配置属性的详细阐述放到第5章。
- static：在这个文件夹下，你可以存放任意为浏览器提供服务的静态内容（图片、样式表、JavaScript等），该文件夹初始为空。
- templates：这个文件夹中存放用来渲染内容到浏览器的模板文件。这个文件夹初始是空的，不过我们很快就会往里面添加Thymeleaf模板。
- TacoCloudApplicationTests.java：这是一个简单的测试类，它能确保Spring应用上下文可以成功加载。在开发应用的过程中，我们会将更多的测试添加进来。

随着Taco Cloud应用功能的增长，我们会不断使用Java代码、图片、样式表、测试以及其他附属内容来充实这个项目结构。不过，在此之前，我们先看一下Spring Initializr提供的几个条目。

探索构建规范

在填充Initializr表单的时候，我们声明项目要使用Maven来进行构建。因此，Spring Initializr所生成的pom.xml文件已经包含了我们所选择的依赖。程序清单 1.1 展示了Initializr为我们提供的完整pom.xml。

程序清单1.1　初始的Maven构建规范

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>sia</groupId>
  <artifactId>taco-cloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>           ←--- 打包为JAR

  <name>taco-cloud</name>
  <description>Taco Cloud Example</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>                   ←--- Spring Boot的
版本
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>
        UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>
        UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>                             ←--- Starter依赖
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
        <scope>test</scope>
    </dependency>

     <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>htmlunit-driver</artifactId>
        <scope>test</scope>
      </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>                           ←--- Spring Boot插件
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

在pom.xml文件中，我们第一个需要注意的地方就是<packaging>。我们选择了将应用构建成一个可执行的JAR文件，而不是WAR文件。这可能是你所做出的最奇怪的选择之一，对Web应用来说尤为如此。毕竟，传统的Java Web应用都是打包成WAR文件，JAR只是用来打包库和较为少见的桌面UI应用的。

打包为JAR文件是基于云思维做出的选择。尽管WAR文件非常适合部署到传统的Java应用服务器上，但对于大多数云平台来说它们并不是理想的选择。有些云平台（比如Cloud Foundry）也能够部署和运行WAR文件，但是所有的Java云平台都能够运行可执行的JAR文件。因

此，Spring Initializr默认会使用基于JAR的打包方式，除非我们明确告诉它采用其他的方式。

如果你想要将应用部署到传统的Java应用服务器上，那么需要选择使用基于WAR的打包方式并要包含一个Web初始化类。在第2章中，我们将会更详细地了解如何构建WAR文件。

接下来，请留意<parent>元素，更具体来说是它的<version>子元素。这表明我们的项目要以spring-boot-starter-parent作为其父POM。除了其他的一些功能之外，这个父POM为Spring项目常用的一些库提供了依赖管理，现在你不需要指定它们的版本，因为这是通过父POM来管理的。这里的2.0.4.RELEASE表明要使用Spring Boot 2.0.4，所以会根据这个版本的Spring Boot定义来继承依赖管理。

既然我们谈到了依赖的话题，那么需要注意在<dependencies>元素下声明了 3 个依赖。在某种程度上，你可能会对前两个更熟悉一些。它们直接对应我们在Spring Tool Suite新项目向导中点击Finish之前所选择的Web和Thymeleaf依赖。第三个依赖提供了很多有用的测试功能。我们没有必要在专门的复选框中选择它，因为Spring Initializr假定你将会编写测试（希望你会正确地开展这项工作）。

你可能也会注意到这 3 个依赖的artifact ID上都有starter这个单词。Spring Boot starter依赖的特别之处在于它们本身并不包含库代码，而是传递性地拉取其他的库。这种starter依赖主要有3个好处。

- 构建文件会显著减小并且更易于管理，因为这样不必为每个所需的

依赖库都声明依赖。

- 我们能够根据它们所提供的功能来思考依赖，而不是根据库的名称。如果是开发Web应用，那么你只需要添加web starter就可以了，而不必添加一堆单独的库再编写Web应用。
- 我们不必再担心库版本的问题。你可以直接相信给定版本的Spring Boot，传递性引入的库的版本是兼容的。现在，你只需要关心使用的是哪个版本的Spring Boot就可以了。

最后，构建规范还包含一个Spring Boot插件。这个插件提供了一些重要的功能。

- 它提供了一个Maven goal，允许我们使用Maven来运行应用。在1.3.4小节，我们将会尝试这个goal。
- 它会确保依赖的所有库都会包含在可执行JAR文件中，并且能够保证它们在运行时类路径下是可用的。
- 它会在JAR中生成一个manifest文件，将引导类（在我们的场景中，也就是TacoCloudApplication）声明为可执行JAR的主类。

谈到了主类，我们打开它看一下。

引导应用

因为我们将会通过可执行JAR文件的形式来运行应用，所以很重要的一点就是要有一个主类，它将会在JAR运行的时候被执行。我们同时还需要一个最小化的Spring配置，以引导该应用。这就是TacoCloudApplication类所做的事情，如程序清单1.2所示。

程序清单1.2　Taco Cloud的引导类

```
package tacos;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication                  ←--- Spring Boot应用
public class TacoCloudApplication {

  public static void main(String[] args) {
    SpringApplication.run(TacoCloudApplication.class, args);    ←--- 运行
应用
  }

}
```

尽管在TacoCloudApplication中只有很少的代码，但是它包含了很多的内容。其中，最强大的一行代码也是最短的。@SpringBootApplication注解明确表明这是一个Spring Boot应用。但是，@SpringBootApplication远比看上去更强大。

@SpringBootApplication是一个组合注解，它组合了3个其他的注解。

- @SpringBootConfiguration：将该类声明为配置类。尽管这个类目前还没有太多的配置，但是后续我们可以按需添加基于Java的Spring框架配置。这个注解实际上是@Configuration注解的特殊形式。
- @EnableAutoConfiguration：启用Spring Boot的自动配置。我们随后会介绍自动配置的更多功能。就现在来说，我们只需要知道这个注解会告诉Spring Boot自动配置它认为我们会用到的组件。
- @ComponentScan：启用组件扫描。这样我们能够通过像@Component、@Controller、@Service这样的注解声明其他类，Spring会自动发现它们并将它们注册为Spring应用上下文中的组件。

TacoCloudApplication另外一个很重要的地方是它的main()方法。这是JAR文件执行的时候要运行的方法。在大多数情况下，这个方法都是样板代码，我们编写的每个Spring Boot应用都会有一个类似或完全相同的方法（类名不同则另当别论）。

这个main()方法会调用SpringApplication中静态的run()方法，后者会真正执行应用的引导过程，也就是创建Spring的应用上下文。在传递给run()的两个参数中，一个是配置类，另一个是命令行参数。尽管传递给run()的配置类不一定要和引导类相同，但这是最便利和最典型的做法。

你可能并不需要修改引导类中的任何内容。对于简单的应用程序来说，你可能会发现在引导类中配置一两个组件是非常方便的，但是对于大多数应用来说，最好还是要为没有实现自动配置的功能创建一个单独的配置类。在本书的整个过程中，我们将会创建多个配置类，所以请继续关注后续的细节。

测试应用

测试是软件开发的重要组成部分。鉴于此，Spring Initializr为我们提供了一个测试类作为起步。程序清单1.3展现了这个测试类的概况。

程序清单1.3　应用测试类的概况

```
package tacos;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```

```
@RunWith(SpringRunner.class)                          ←--- 使用Spring的运行器
@SpringBootTest                                   ←--- Spring Boot测试
public class TacoCloudApplicationTests {

  @Test                             ←--- 测试方法
  public void contextLoads() {
  }

}
```

TacoCloudApplicationTests类中的内容并不多：这个类中只有一个空的测试方法。即便如此，这个测试类还是会执行必要的检查，确保Spring应用上下文能够成功加载。如果你所做的变更导致Spring应用上下文无法创建，那么这个测试将会失败，你就可以做出反应来解决相关的问题了。

另外，注意这个类带有@RunWith(SpringRunner.class)注解。@RunWith是JUnit的注解，它会提供一个测试运行器（runner）来指导JUnit如何运行测试。可以将其想象为给JUnit应用一个插件，以提供自定义的测试行为。在本例中，为JUnit提供的是SpringRunner，这是一个Spring提供的测试运行器，它会创建测试运行所需的Spring应用上下文。

测试运行器的其他名称

如果你已经熟悉如何编写Spring测试或者见过其他一些基于Spring的测试类，那么你可能见过名为SpringJUnit4ClassRunner的测试运行器。SpringRunner是SpringJUnit4ClassRunner的别名，是在Spring 4.3中

引入的，以便于移除对特定JUnit版本的关联（比如，JUnit 4）。毫无疑问，这个别名更易于阅读和输入。

@SpringBootTest会告诉JUnit在启动测试的时候要添加上Spring Boot的功能。从现在开始，我们可以将这个测试类视同为在main()方法中调用SpringApplication.run()。在这本书中，我们将会多次看到@SpringBootTest，而且会不断见识它的威力。

最后，就是测试方法本身了。尽管@RunWith(SpringRunner.class)和@SpringBootTest会为测试加载Spring应用上下文，但是如果没有任何测试方法，那么它们其实什么事情都没有做。即便没有任何断言或代码，这个空的测试方法也会提示这两个注解完成了它们的工作并成功加载Spring应用上下文。如果这个过程中有任何问题，那么测试都会失败。

此时，我们已经看完了Spring Initializr为我们提供的代码。我们看到了一些用来开发Spring应用程序的基础样板，但是还没有编写任何代码。现在是时候启动IDE、准备好键盘并向Taco Cloud应用程序添加一些自定义的代码了。

1.3　编写Spring应用

因为是刚刚开始，所以我们首先为Taco Cloud做一些小的变更，但是这些变更会展现Spring的很多优点。在刚开始的时候，比较合适的做法是为Taco Cloud应用添加一个主页。在添加主页时，我们将会创建两个代码构件:

- 一个控制器类，用来处理主页相关的请求；
- 一个视图模板，用来定义主页看起来是什么样子。

测试是非常重要的，所以我们还会编写一个简单的测试类来测试主页。但是，要事优先，我们需要先编写控制器。

1.3.1 处理Web请求

Spring自带了一个强大的Web框架，名为Spring MVC。Spring MVC的核心是控制器（controller）的理念。控制器是处理请求并以某种方式进行信息响应的类。在面向浏览器的应用中，控制器会填充可选的数据模型并将请求传递给一个视图，以便于生成返回给浏览器的HTML。

在第2章中，我们将会学习更多关于Spring MVC的知识。现在，我们会编写一个简单的控制器类以处理对根路径（比如，“/”）的请求，并将这些请求转发至主页视图，在这个过程中不会填充任何的模型数据。程序清单 1.4 展示了这个简单的控制器类。

程序清单1.4　主页控制器

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller                                ←--- 控制器
public class HomeController {

  @GetMapping("/")                         ←--- 处理对根路径“/”的请求
  public String home() {
    return "home";          ←--- 返回视图名
  }
```

```
}
```

可以看到，这个类带有@Controller。就其本身而言，@Controller并没有做太多的事情。它的主要目的是让组件扫描将这个类识别为一个组件。因为HomeController带有@Controller，所以Spring的组件扫描功能会自动发现它，并创建一个HomeController实例作为Spring应用上下文中的bean。

实际上，有一些其他的注解与@Controller有着类似的目的（包括@Component、@Service和@Repository）。你可以为HomeController添加上述的任意其他注解，其作用是完全相同的。但是，在这里选择使用@Controller更能描述这个组件在应用中的角色。

home()是一个简单的控制器方法。它带有@GetMapping注解，表明如果针对“/”发送HTTP GET请求，那么这个方法将会处理请求。该方法所做的只是返回String类型的home值。

这个值将会被解析为视图的逻辑名。视图如何实现取决于多个因素，但是因为Thymeleaf位于类路径中，所以我们可以使用Thymeleaf来定义模板。

为何使用**Thymeleaf**

你可能会想为什么要选择Thymeleaf作为模板引擎呢？为何不使用JSP？为何不使用FreeMarker？为何不选择其他的几个可选方案？

简单来说，我必须要做出选择，我喜欢Thymeleaf，相对于其他的方案，我会优先使用它。即便JSP是更加显而易见的选择，但是组合使用JSP和Spring Boot需要克服一些挑战。我不想脱离第1章的内容定位，所以在这里就此打住。在第2章中，我们将会看一下其他的模板方案，其中也包括JSP。

模板名称是由逻辑视图名派生而来的，再加上“/templates/”前缀和“.html”后缀。最终形成的模板路径将是“/templates/home.html”。所以，我们需要将模板放到项目的“/src/main/resources/templates/home.html”目录中。现在，就让我们来创建这个模板。

1.3.2　定义视图

为了让主页尽可能简单，除了欢迎用户访问站点之外，它不会做其他的任何事情。程序清单1.5展现了基本的Thymeleaf模板，它定义了Taco Cloud的主页。

程序清单1.5　Taco Cloud主页模板

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Welcome to...</h1>
    <img th:src="@{/images/TacoCloud.png}"/>
```

```
  </body>
</html>
```

这个模板并没有太多需要讨论的。唯一需要注意的一行代码是用于展现Taco Cloud Logo的<img>标签。它使用了Thymeleaf的th:src属性和@{...}表达式，以便于引用相对于上下文路径的图片。除此之外，它就是一个Hello World页面。

但是，我们再讨论一下这个图片。我将定义Taco Cloud Logo的工作留给你，你需要将它放到应用的正确位置中。

图片是使用相对于上下文的“/images/TacoCloud.png”路径来进行引用的。回忆一下我们的项目结构，像图片这样的静态资源是放到“/src/main/resources/static”文件夹中的。这意味着，在项目中，Taco Cloud Logo图片必须要位于“/src/main/resources/static/images/TacoCloud.png”。

我们已经有了一个处理主页请求的控制器并且有了渲染主页的模板，现在基本就可以启动应用来看一下它的效果了。在此之前，我们先看一下如何为控制器编写测试。

1.3.3 测试控制器

在测试Web应用时，对HTML页面的内容进行断言是比较困难的。幸好Spring对测试提供了强大的支持，这使得测试Web应用变得非常简单。

对于主页来说，我们所编写的测试在复杂性上与主页本身差不多。测试需要针对根路径“/”发送一个HTTP GET请求并期望得到成功结果，其中视图名称为home并且结果内容包含“Welcome to...”。程序清单1.6能够完成该任务。

程序清单1.6　针对主页控制器的测试

```
package tacos;

import static org.hamcrest.Matchers.containsString;
import static
     org.springframework.test.web.servlet.request.MockMvcRequestBuilders.g
et;
import static
     org.springframework.test.web.servlet.result.MockMvcResultMatchers.con
tent;
import static
     org.springframework.test.web.servlet.result.MockMvcResultMatchers.sta
tus;
import static
     org.springframework.test.web.servlet.result.MockMvcResultMatchers.vie
w;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@RunWith(SpringRunner.class)
@WebMvcTest(HomeController.class)         ←--- 针对HomeController的Web测试
public class HomeControllerTest {

  @Autowired
  private MockMvc mockMvc;                  ←--- 注入MockMvc

  @Test
  public void testHomePage() throws Exception {
    mockMvc.perform(get("/"))                 ←--- 发起对“/”的GET

      .andExpect(status().isOk())             ←--- 期望得到HTTP 200
```

```
        .andExpect(view().name("home"))         ←--- 期望得到home视图

        .andExpect(content().string(           ←--- 期望包含“Welcome to...
”
               containsString("Welcome to...")));
  }

}
```

对于这个测试，我们首先注意到的可能就是它使用了与TacoCloudApplicationTests类不同的注解。HomeControllerTest没有使用@SpringBootTest标记，而是添加了@WebMvcTest注解。这是Spring Boot所提供的一个特殊测试注解，它会让这个测试在Spring MVC应用的上下文中执行。更具体来讲，在本例中，它会将HomeController注册到Spring MVC中，这样的话，我们就可以向它发送请求了。

@WebMvcTest同样会为测试Spring MVC应用提供Spring环境的支持。尽管我们可以启动一个服务器来进行测试，但是对于我们的场景来说，仿造一下Spring MVC的运行机制就可以。测试类被注入了一个MockMvc，能够让测试实现mockup。

通过testHomePage()方法，我们定义了针对主页想要执行的测试。它首先使用MockMvc对象对“/”（根路径）发起HTTP GET请求。对于这个请求，我们设置了如下的预期：

- 响应应该具备HTTP 200 (OK)状态；
- 视图的逻辑名称应该是home；
- 渲染后的视图应该包含文本“Welcome to...”。

如果在MockMvc对象发送请求之后，这些期望有不满足的话，那么

这个测试会失败。但是，我们的控制器和模板引擎在编写时都满足了这些预期，所以测试应该能够通过，并且带有成功的图标——至少能够看到一些绿色的背景，表明测试通过了。

控制器已经编写好了，视图模板也已经创建完毕，而且我们还通过了测试，看上去我们已经成功实现了主页。尽管测试已经通过了，但是如果能够在浏览器中看到结果那会更有成就感，毕竟这才是Taco Cloud的客户所能看到的效果。接下来，我们构建应用并运行它。

1.3.4　构建和运行应用

就像初始化Spring应用有多种方式一样，运行Spring应用也有多种方式。如果你愿意的话，可以翻到附录部分，以了解运行Spring Boot应用的一些通用方式。

因为我们选择了使用Spring Tool Suite来初始化和处理项目，所以可以借助名为Spring Boot Dashboard的便捷功能来帮助我们在IDE中运行应用。Spring Boot Dashboard的表现形式是一个Tab标签，通常会位于IDE窗口的左下角附近。图1.7展现了一个带有标注的Spring Boot Dashboard截屏。

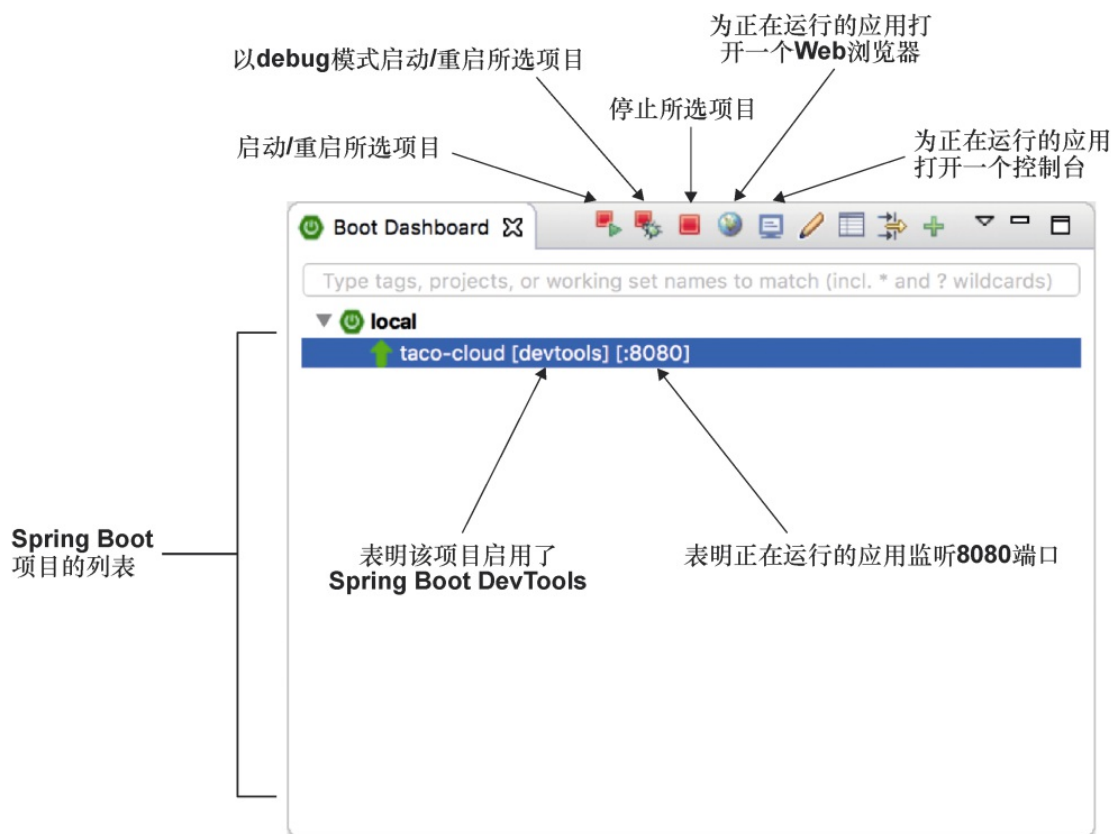


图1.7　Spring Boot Dashboard的重点功能

图1.7包含了一些最有用的细节，但是我不想花太多时间介绍Spring Boot Dashboard支持的所有功能。对我们来说，现在最重要的事情是需要知道如何使用它来运行TacoCloud应用。确保taco-cloud应用程序在项目列表中能够显示出来（这是图1.7中显示的唯一应用），然后点击启动按钮（上方工具栏最左边的按钮，也就是带有绿色三角形和红色正方形的按钮），应用程序应该就能立即启动。

在应用启动的过程中，你会在控制台看到一些Spring ASCII码，随后会是描述应用启动各个步骤的日志条目。在控制台输出的最后，你将会看到一条Tomcat已经在port(s): 8080 (http)启动的日志，这意味着此时你可以打开Web浏览器并导航至主页，这样就能看到我们的劳动成果

了。

稍等一下！刚才说启动Tomcat？但是我们是什么时候将应用部署到Tomcat的呢？

Spring Boot应用的习惯做法是将所有它需要的东西都放到一起，没有必要将其部署到某种应用服务器中。在这个过程中，我们根本没有将应用部署到Tomcat中……Tomcat是我们应用的一部分！（在1.3.6小节，我会介绍Tomcat是如何成为我们应用的一部分的。）

现在，应用已经启动起来了，打开Web浏览器并访问http://localhost:8080（或者在Spring Boot Dashboard中点击上方的地球样式的按钮，如图1.7所示），你将会看到如图1.8所示的界面。如果你设计了自己的Logo图片，那么显示效果可能会有所不同。但是，与图1.8相比，应该不会有太大的差异。

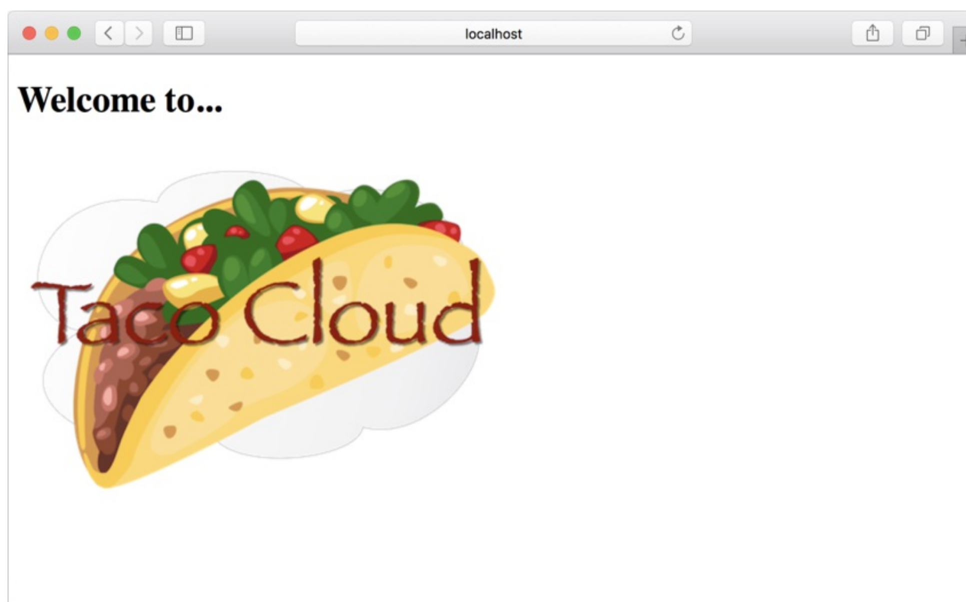


图1.8　Taco Cloud主页

看上去似乎并不太美观，但这不是一本关于平面设计的书。目前，略显简陋的主页外观已经足够了，它为我们学习Spring打下了一个良好的开端。

到现在为止，我一直没有提及DevTools。在初始化项目的时候，我们将其作为一个依赖添加了进来。在最终生成的pom.xml文件中，它表现为一个依赖项。甚至Spring Boot Dashboard都显示项目启用了DevTools。那么，DevTools是什么，它又能为我们做些什么呢？接下来，让我们快速浏览一下DevTools最有用的一些特性。

1.3.5 了解Spring Boot DevTools

顾名思义，DevTools为Spring开发人员提供了一些便利的开发期工具，其中包括：

- 代码变更后应用会自动重启；
- 当面向浏览器的资源（如模板、JavaScript、样式表）等发生变化时，会自动刷新浏览器；
- 自动禁用模板缓存；
- 如果使用H2数据库的话，内置了H2控制台。

需要注意，DevTools并不是IDE插件，它也不需要你使用特定的IDE。在Spring Tool Suite、IntelliJ IDEA和NetBeans中，它都能很好地运行。另外，因为它的目的是仅仅用于开发，所以能够很智能地在生产环境中把自己禁用掉。（我们将会在第19章学习应用部署的时候再讨论它是如何做到这一点的。）现在，我们主要关注Spring Boot DevTools最有

用的特性，先从应用的自动重启开始。

应用自动重启

如果将DevTools作为项目的一部分，那么你可以看到，当对项目中的Java代码和属性文件做出修改后，这些变更稍后就能发挥作用。DevTools会监控变更，当它看到有变化的时候，将会自动重启应用。

更准确地说，当DevTools运行的时候，应用程序会被加载到Java虚拟机（Java virtual Machine，JVM）两个独立的类加载器中。其中一个类加载器会加载你的Java代码、属性文件以及项目中“src/main/”路径下几乎所有的内容。这些条目很可能会经常发生变化。另外一个类加载器会加载依赖的库，这些库不太可能经常发生变化。

当探测到变更的时候，DevTools只会重新加载包含项目代码的类加载器，并重启Spring的应用上下文，在这个过程中另外一个类加载器和JVM会原封不动。这个策略非常精细，但是它能减少应用启动的时间。

这种策略的一个不足之处就是自动重启无法反映依赖项的变化。这是因为包含依赖库的类加载器不会自动重新加载。这意味着每当我们在构建规范中添加、变更或移除依赖的时候，为了让变更生效，我们需要重新启动应用。

浏览器自动刷新和禁用模板缓存

默认情况下，像Thymeleaf和FreeMarker这样的模板方案在配置时会

缓存模板解析的结果。这样的话，在为每个请求提供服务的时候，模板就不用重新解析了。在生产环境中，这是一种很好的方式，因为它会带来一定的性能收益。

但是，在开发期，缓存模板就不太好了。在应用运行的时候，如果缓存模板，那么我们刷新浏览器就无法看到模板变更的效果了。即便我们对模板做了修改，在应用重启之前，缓存的模板依然会有效。

DevTools通过禁用所有模板缓存解决了这个问题。你可以对模板进行任意数量的修改，只需要刷新一下浏览器就能看到结果。

如果你像我这样，连浏览器的刷新按钮都懒得点，那么对代码做出变更之后，马上在浏览器中看到结果就好了。幸运的是，DevTools有一些特殊的功能可以供我们使用。

DevTools在运行的时候，它会和你的应用程序一起，同时自动启动一个LiveReload服务器。LiveReload服务器本身并没有太大的用处。但是，当它与LiveReload浏览器插件结合起来的时候，就能够在模板、图片、样式表、JavaScript等（实际上，几乎涵盖为浏览器提供服务的所有内容）发生变化的时候自动刷新浏览器。

LiveReload有针对Google Chrome、Safari和Firefox的浏览器插件（要对Internet Explorer和Edge粉丝说声抱歉）。请访问LiveReload官网，以了解如何为你的浏览器安装LiveReload。

内置的H2控制台

虽然我们的项目还没有使用数据库，但是这种情况在第3章中就会发生变化。如果你使用H2数据库进行开发，DevTools将会自动启用H2。这样的话，我们可以通过Web浏览器进行访问。你只需要让浏览器访问http://localhost:8080/h2-console，就能看到应用所使用的数据。

此时，我们已经编写了一个尽管非常简单却很完整的Spring应用。在本书中，我们将会不断扩展它。现在，我们要回过头来看一下都完成了哪些工作以及Spring发挥了什么作用。

1.3.6 回顾一下

回想一下我们是怎样完成这一切的。简短来说，在构建基于Spring的Taco Cloud应用的过程中，我们执行了如下步骤：

- 使用Spring Initializr创建初始的项目结构；
- 编写控制器类处理针对主页的请求；
- 定义了一个视图模板来渲染主页；
- 编写了一个简单的测试类来验证工作符合预期。

这些步骤都非常简单直接，对吧？除了初始化应用的第一个步骤之外，我们所做的每一个操作都专注于生成主页的目标。

实际上，我们所编写的每行代码都致力于实现这个目标。除了Java import语句之外，我只能在控制器中找到两行Spring相关的代码，而在视图模板中一行Spring相关的代码都没有。尽管测试类的大部分内容都使用了Spring对测试的支持，但是它在测试的上下文中似乎没有那么具

有侵入性。

这是使用Spring进行开发的一个重要收益。你可以只关注满足应用需求的代码，无须考虑如何满足框架的需求。尽管我们偶尔还是需要编写一些框架特定的代码，但是它们通常只占整个代码库很小的一部分。正如我在前文所述，Spring（以及Spring Boot）可以视为感受不到框架的框架（frameworkless framework）。

但是这又是如何运行起来的呢？Spring在幕后做了些什么来保证应用的需求能够得到满足呢？要理解Spring到底做了些什么，我们首先来看一下构建规范。

在pom.xml文件中，我们声明了对Web和Thymeleaf starter的依赖。这两项依赖会传递引入大量其他的依赖，包括：

- Spring的MVC框架；
- 嵌入式的Tomcat；
- Thymeleaf和Thymeleaf布局方言；

它还引入了Spring Boot的自动配置库。当应用启动的时候，Spring Boot的自动配置将会探测到这些库，并自动完成如下功能：

- 在Spring应用上下文中配置bean以启用Spring MVC；
- 在Spring应用上下文中配置嵌入式的Tomcat服务器；
- 配置Thymeleaf视图解析器，以便于使用Thymeleaf模板渲染Spring MVC视图。

简而言之，自动配置功能完成了所有的脏活累活，让我们能够集中

精力编写实现应用功能的代码。如果你问我对此的观点，那么我认为这是一个很好的安排！

我们的Spring之旅才刚刚开始。Taco Cloud应用程序只涉及Spring所提供功能的一小部分。在开始下一步之前，我们先整体了解一下Spring，看看在我们的路途中都会有哪些地标。

1.4 俯瞰Spring风景线

要想了解Spring的整体状况，只需查看完整版本的Spring Initializr Web表单上的那一堆复选框列表即可。它列出了100多个可选的依赖项，所以我不会在这里列出所有选项，也不会提供截图，但我鼓励你去看一看。同时，在这里我会简单介绍一些重点的项目。

1.4.1 Spring核心框架

如你所料，Spring核心框架是Spring领域中一切的基础。它提供了核心容器和依赖注入框架，另外还提供了一些其他重要的特性。

其中有一项是Spring MVC，也就是Spring的Web框架。你已经看到了如何使用Spring MVC来编写控制器类以处理Web请求。但是，你还没看到的是，Spring MVC还能用来创建REST API，以生成非HTML的输出。在第2章中，我们将会更深入地介绍Spring MVC，并在第6章重新学习如何使用它来创建REST API。

Spring核心框架还提供了一些对数据持久化的基础支持，尤其是基于模板的JDBC支持。在第3章中，你将会看到如何使用JdbcTemplate。

在最新版本的Spring中，还添加了对反应式（reactive）风格编程的支持，其中包括名为Spring WebFlux的新反应式Web框架，这个框架大量借鉴了Spring MVC。在第3部分中，我们将会学习Spring反应式编程模型，并在第11章专门学习Spring WebFlux。

1.4.2 Spring Boot

我们已经看到了Spring Boot带来的很多收益，包括starter依赖和自动配置。在本书中，我们会尽可能多地使用Spring Boot，并避免任何形式的显式配置，除非显式配置是绝对必要的。除了starter依赖和自动配置，Spring Boot还提供了大量其他有用的特性：

- Actuator能够洞察应用运行时的内部工作状况，包括指标、线程dump信息、应用的健康状况以及应用可用的环境属性；
- 灵活的环境属性规范；
- 在核心框架的测试辅助功能之上提供了对测试的额外支持。

除此之外，Spring Boot还提供了一个基于Groovy脚本的编程模型，称为Spring Boot 命令行接口（Command-Line Interface，CLI）。使用Spring Boot CLI，我们可以将整个应用程序编写为Groovy脚本的集合，并通过命令行运行它们。我们不会花太多时间介绍Spring Boot CLI，但是当它匹配我们的需求时，我们会偶尔提及它。

Spring Boot已经成为Spring开发中不可或缺的一部分，很难想象如果没有它我该如何开发Spring应用程序。因此，本书采用以Spring Boot为核心的视角。当我介绍Spring Boot所做的事情时，你可能会发现我却使用了Spring这个词。

1.4.3　Spring Data

尽管Spring核心框架提供了基本的数据持久化支持，但是Spring Data提供了非常令人惊叹的功能：将应用程序的数据repository定义为简单的Java接口，在定义驱动存储和检索数据的方法时使用一种命名约定即可。

此外，Spring Data能够处理多种不同类型的数据库，包括关系型数据库（JPA）、文档数据库（Mongo）、图数据库（Neo4j）等。在第3章中，我们将使用Spring Data为Taco Cloud应用程序创建repository。

1.4.4　Spring Security

应用程序的安全性一直是一个重要的话题，而且正在变得越来越重要。幸运的是，Spring有一个健壮的安全框架，名为Spring Security。

Spring Security解决了应用程序通用的安全性需求，包括身份验证、授权和API安全性。Spring Security的范围太大，在本书中无法得到充分的介绍，但是我们将在第4章和第11章中讨论一些常见的使用场

景。

1.4.5　Spring Integration和Spring Batch

从一定程度上来讲，大多数应用程序都需要与其他应用甚至本应用中的其他组件进行集成。在这方面，有一些应用程序集成模式可以解决这些需求。Spring Integration和Spring Batch为基于Spring的应用程序提供了这些模式的实现。

Spring Integration解决了实时集成问题。在实时集成中，数据在可用时马上就会得到处理。相反，Spring Batch解决的则是批处理集成的问题，在此过程中，数据可以收集一段时间，直到某个触发器（可能是一个时间触发器）发出信号，表示该处理批量数据了才会对数据进行批处理。我们将会在第9章中研究Spring Batch和Spring Integration。

1.4.6　Spring Cloud

在撰写本书的时候，应用程序开发领域正在进入一个新的时代，我们不再将应用程序作为单个部署单元来开发，而是使用由微服务组成的多个独立部署单元来组合形成应用程序。

微服务是一个热门话题，解决了开发期和运行期的一些实际问题。然而，在这样做的过程中，它们也面临着自己所带来的挑战。这些挑战将由Spring Cloud直面解决，Spring Cloud是使用Spring开发云原生应用

程序的一组项目。

Spring Cloud覆盖了很多领域，本书不可能面面俱到，我们将在第13～15章中研究Spring Cloud的一些常见组件。要更全面地研究Spring Cloud，我建议阅读John Carnell的*Spring Microservices in Action*一书[2]（Manning，2017）。

1.5　小结

- Spring旨在简化开发人员所面临的挑战，比如创建Web应用程序、处理数据库、保护应用程序以及实现微服务。
- Spring Boot构建在Spring之上，通过简化依赖管理、自动配置和运行时洞察，使Spring更加易用。
- Spring应用程序可以使用Spring Initializr进行初始化。Spring Initializr是基于Web的应用，并且为大多数Java开发环境提供了原生支持。
- 在Spring应用上下文中，组件（通常称为bean）既可以使用Java或XML显式声明，也可以通过组件扫描发现，还可以使用Spring Boot自动配置功能实现自动化配置。

---

[1]　为了行文简洁，同时保持与示例应用中Web页面展现的一致性，我们后文不再将taco翻译为墨西哥煎玉米卷，而是直接使用taco这一叫法。——译者注

[2]　该书中文版《Spring微服务实战》已由人民邮电出版社出版（ISBN978-7-115-48118-4）。

第2章　开发Web应用

本章内容：

- 在浏览器中展现模型数据
- 处理和校验表单输入
- 选择视图模板库

第一印象是非常重要的：Curb Appeal能够在购房者真正进门之前就将房子卖掉；如果一辆车喷成了樱桃色，那么它的油漆会比它的引擎更引人注目；文学作品中充满了一见钟情的故事。内在固然非常重要，但是外在的，也就是第一眼看到的东西同样非常重要。

我们使用Spring所构建的应用能完成各种各样的事情，包括处理数据、从数据库中读取信息以及与其他应用进行交互。但是，用户对应用程序的第一印象来源于用户界面。在很多应用中，UI是以浏览器中的Web应用的形式来展现的。

在第1章中，我们创建了第一个Spring MVC控制器来展现应用的主

页。但是，Spring MVC能做很多的事情，并不局限于展现静态内容。在本章中，我们将会开发Taco Cloud的第一个主要功能：设计定制taco的能力。在这个过程中，我们将会深入研究Spring MVC并会看到如何展现模型数据和处理表单输入。

2.1　展现信息

从根本上来讲，Taco Cloud是一个可以在线订购taco的地方。但是，除此之外，Taco Cloud允许客户展现其创意，能够让他们通过丰富的配料（ingredient）设计自己的taco。

因此，Taco Cloud需要有一个页面为taco艺术家展现都可以选择哪些配料。可选的配料可能随时会发生变化，所以不能将它们硬编码到HTML页面中。我们应该从数据库中获取可用的配料并将其传递给页面，进而展现给客户。

在Spring Web应用中，获取和处理数据是控制器的任务，而将数据渲染到HTML中并在浏览器中展现则是视图的任务。为了支撑taco的创建页面，我们需要构建如下组件。

- 用来定义taco配料属性的领域类。
- 用来获取配料信息并将其传递至视图的Spring MVC控制器类。
- 用来在用户的浏览器中渲染配料列表的视图模板。

这些组件之间的关系如图2.1所示。

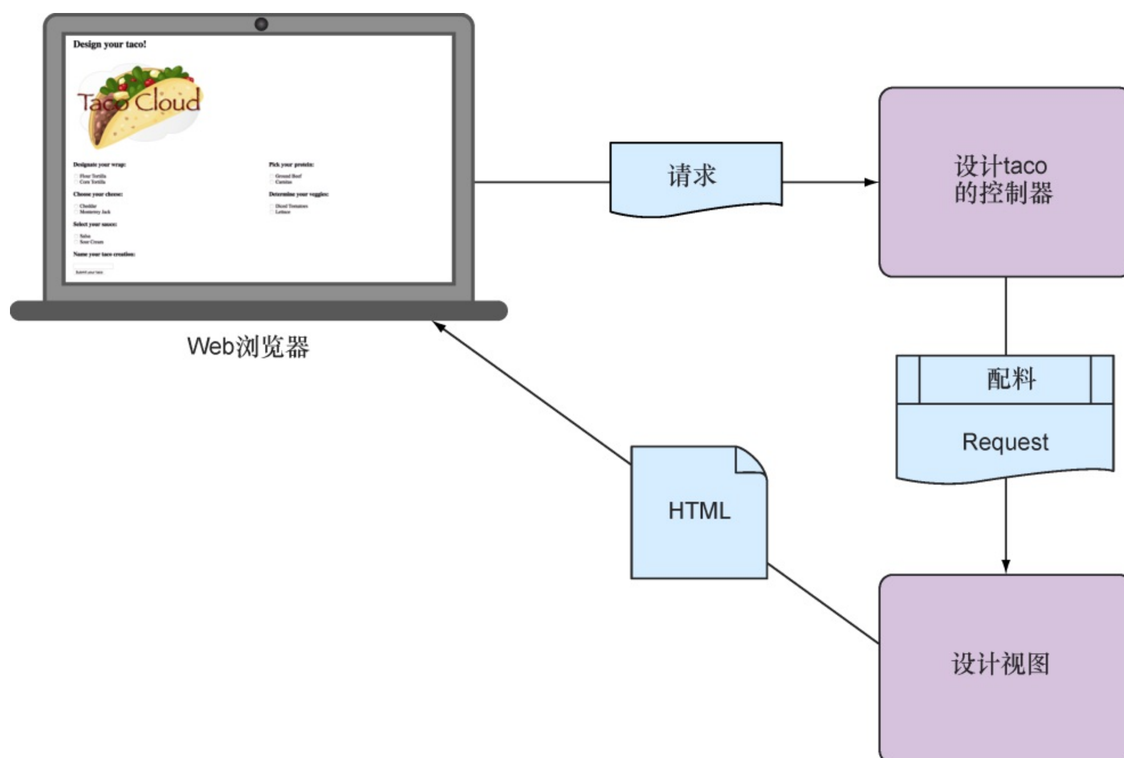


图2.1　典型的Spring MVC请求流

因为本章主要关注Spring的Web框架，所以我们会将数据库相关的内容放到第3章中进行讲解。现在的控制器只负责向视图提供配料。在第3章中，我们会重新改造这个控制器，让它能够与repository协作，从数据库中获取配料数据。

在编写控制器和视图之前，我们首先确定一下用来表示配料的领域类型，它会为我们开发Web组件奠定基础。

2.1.1　构建领域类

应用的领域指的是它所要解决的主题范围：也就是会影响到对应用理解的理念和概念[1]。在Tao Cloud应用中，领域对象包括taco设计、组

成这些设计的配料、顾客以及顾客所下的订单。作为开始，我们首先关注taco的配料。

在我们的领域中，taco配料是非常简单的对象。每种配料都有一个名称和类型，以便于对其进行可视化的分类（蛋白质、奶酪、酱汁等）。每种配料还有一个ID，这样的话对它的引用就能非常容易和明确。如下的Ingredient类定义了我们所需的领域对象。

程序清单2.1　定义taco配料

```
package tacos;

import lombok.Data;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Ingredient {

  private final String id;
  private final String name;
  private final Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

我们可以看到，这是一个非常普通的Java领域类，它定义了描述配料所需的3个属性。在程序清单2.1中，Ingredient类最不寻常的一点是它似乎缺少了常见的getter和setter方法，以及equals()、hashCode()、toString()等方法。

在程序清单2.1中没有这些方法，部分原因是节省空间，此外还因

为我们使用了名为Lombok的库（这是一个非常棒的库，它能够在运行时动态生成这些方法）。实际上，类级别的@Data注解就是由Lombok提供的，它会告诉Lombok生成所有缺失的方法，同时还会生成所有以final属性作为参数的构造器。通过使用Lombok，我们能够让Ingredient的代码简洁明了。

Lombok并不是Spring库，但是它非常有用，我发现如果没有它，开发工作将很难开展。当我需要在书中将代码示例编写得短小简洁时，它简直成了我的救星。

要使用Lombok，首先要将其作为依赖添加到项目中。如果你使用Spring Tool Suite，那么只需要用右键点击pom.xml，并从Spring上下文菜单选项中选择“Edit Starters”。在第1章中看到的选择依赖的对话框将会再次出现（见图1.4），这样的话我们就有机会添加依赖或修改已选择的依赖了。找到Lombok选项，并确保它处于已选中的状态，然后点击“OK”，Spring Tool Suite会自动将其添加到构建规范中。

另外，你也可以在pom.xml中通过如下条目进行手动添加：

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

这个依赖将会在开发阶段为你提供Lombok注解（例如@Data），并且会在运行时进行自动化的方法生成。但是，我们还需要将Lombok作为扩展添加到IDE上，否则IDE将会报错，提示缺少方法和final属性没有

赋值。参见Lombok项目页面，以查阅如何在你所选择的IDE上安装Lombok。

我相信你会发现Lombok非常有用，但你也要知道，它是可选的。在开发Spring应用的时候，它并不是必备的，所以如果你不想使用它的话，完全可以手动编写这些缺失的方法。当你完成之后，我们将会在应用中添加一些控制器，让它们来处理Web请求。

2.1.2　创建控制器类

在Spring MVC框架中，控制器是重要的参与者。它们的主要职责是处理HTTP请求，要么将请求传递给视图以便于渲染HTML（浏览器展现），要么直接将数据写入响应体（RESTful）。在本章中，我们将会关注使用视图来为Web浏览器生成内容的控制器。在第6章中，我们将会看到如何以REST API的形式编写控制器来处理请求。

对于Taco Cloud应用来说，我们需要一个简单的控制器，它要完成如下功能。

- 处理路径为“/design”的HTTP GET请求。
- 构建配料的列表。
- 处理请求，并将配料数据传递给要渲染为HTML的视图模板，发送给发起请求的Web浏览器。

程序清单2.2中的DesignTacoController类解决了这些需求。

程序清单2.2　初始的Spring控制器类

```
package tacos.web;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.slf4j.Slf4j;
import tacos.Taco;
import tacos.Ingredient;
import tacos.Ingredient.Type;

@Slf4j
@Controller
@RequestMapping("/design")
public class DesignTacoController {

  @GetMapping
  public String showDesignForm(Model model) {
    List<Ingredient> ingredients = Arrays.asList(
      new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
      new Ingredient("COTO", "Corn Tortilla", Type.WRAP),
      new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
      new Ingredient("CARN", "Carnitas", Type.PROTEIN),
      new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),
      new Ingredient("LETC", "Lettuce", Type.VEGGIES),
      new Ingredient("CHED", "Cheddar", Type.CHEESE),
      new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),
      new Ingredient("SLSA", "Salsa", Type.SAUCE),
      new Ingredient("SRCR", "Sour Cream", Type.SAUCE)
    );

    Type[] types = Ingredient.Type.values();
    for (Type type : types) {
      model.addAttribute(type.toString().toLowerCase(),
          filterByType(ingredients, type));
    }

    model.addAttribute("design", new Taco());
```

```
    return "design";
  }
  private List<Ingredient> filterByType(
      <Ingredient> ingredients, Type type) {
      return ingredients
         .stream()
         .filter(x -> x.getType().equals(type))
         .collect(Collectors.toList());
  }

}
```

对于DesignTacoController，我们先要注意在类级别所应用的注解。首先是@Slf4j，这是Lombok所提供的注解，在运行时，它会在这个类中自动生成一个SLF4J（Simple Logging Facade for Java）Logger。这个简单的注解和在类中通过如下代码显式声明的效果是一样的：

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(DesignTacoController.class);
```

随后，我们将会用到这个Logger。

DesignTacoController用到的下一个注解是@Controller。这个注解会将这个类识别为控制器，并且将其作为组件扫描的候选者，所以Spring会发现它并自动创建一个DesignTacoController实例，并将该实例作为Spring应用上下文中的bean。

DesignTacoController还带有@RequestMapping注解。当@RequestMapping注解用到类级别的时候，它能够指定该控制器所处理的请求类型。在本例中，它规定DesignTacoController将会处理路径以“/design”开头的请求。

处理GET请求

修饰showDesignForm()方法的@GetMapping注解对类级别的@RequestMapping进行了细化。@GetMapping结合类级别的@RequestMapping，指明当接收到对“/design”的HTTP GET请求时，将会调用showDesignForm()来处理请求。

@GetMapping是一个相对较新的注解，是在Spring 4.3引入的。在Spring 4.3之前，你可能需要使用方法级别的@RequestMapping注解作为替代：

```
@RequestMapping(method=RequestMethod.GET)
```

显然，@GetMapping更加简洁，并且指明了它的目标HTTP方法。@GetMapping只是诸多请求映射注解中的一个。表2.1列出了Spring MVC中所有可用的请求映射注解。

表**2.1　Spring MVC**的请求映射注解

注解	描述
@RequestMapping	通用的请求处理
@GetMapping	处理HTTP GET请求
@PostMapping	处理HTTP POST请求

@PutMapping	处理HTTP PUT请求
@DeleteMapping	处理HTTP DELETE请求
@PatchMapping	处理HTTP PATCH请求

让正确的事情变得更容易

在为控制器方法声明请求映射时，越具体越好。这意味着至少要声明路径（或者从类级别的@RequestMapping继承一个路径）以及它所处理的HTTP方法。

但是更长的@RequestMapping(method=RequestMethod.GET)注解很容易让开发人员采取懒惰的方式，也就是忽略掉method属性。幸亏有了Spring 4.3的新注解，正确的事情变得更容易了，我们的输入变得更少了。

新的请求映射注解具有和@RequestMapping完全相同的属性，所以我们可以在使用@RequestMapping的任何地方使用它们。

通常，我喜欢只在类级别上使用@RequestMapping，以便于指定基本路径。在每个处理器方法上，我会使用更具体的@GetMapping、@PostMapping等注解。

现在，我们已经知道showDesignForm()方法会处理请求，接下来我们看一下方法体，看它都做了些什么工作。这个方法构建了一个Ingredient对象的列表。现在，这个列表是硬编码的。当我们学习第3章的时候，会从数据库中获取可用taco配料并将其放到列表中。

配料列表准备就绪之后，showDesignForm()方法接下来的几行代码会根据配料类型过滤列表。配料类型的列表会作为属性添加到Model对象上，这个对象是以参数的形式传递给showDesignForm()方法的。Model对象负责在控制器和展现数据的视图之间传递数据。实际上，放到Model属性中的数据将会复制到Servlet Response的属性中，这样视图就能在这里找到它们了。showDesignForm()方法最后返回“design”，这是视图的逻辑名称，会用来将模型渲染到视图上。

我们的DesignTacoController已经具备雏形了。如果你现在运行应用并在浏览器上访问“/design”路径，DesignTacoController的showDesignForm()将会被调用，它会从repository中获取数据并放到模型中，然后将请求传递给视图。但是，我们现在还没有定义视图，请求将会遇到很糟糕的问题，也就是HTTP 404（Not Found）。为了解决这个问题，我们将注意力切换到视图上，在这里数据将会使用HTML进行装饰，以便于在用户的Web浏览器中进行展现。

2.1.3 设计视图

在控制器完成它的工作之后，现在就该视图登场了。Spring提供了

多种定义视图的方式，包括JavaServer Pages（JSP）、Thymeleaf、FreeMarker、Mustache和基于Groovy的模板。就现在来讲，我们会使用Thymeleaf，这也是我们在第1章开启这个项目时的选择。我们会在第2.5节考虑其他的可选方案。

为了使用Thymeleaf，我们需要添加另外一个依赖到项目构建中。如下的<dependency>条目使用了Spring Boot的Thymeleaf starter，从而能够让Thymeleaf 渲染我们将要创建的视图：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

在运行时，Spring Boot的自动配置功能会发现Thymeleaf在类路径中，因此会为Spring MVC创建支撑Thymeleaf视图的bean。

像Thymeleaf这样的视图库在设计时是与特定的Web框架解耦的。这样的话，它们无法感知Spring的模型抽象，因此无法与控制器放到Model中的数据协同工作。但是，它们可以与Servlet的request属性协作。所以，在Spring将请求转移到视图之前，它会把模型数据复制到request属性中，Thymeleaf和其他的视图模板方案就能访问到它们了。

Thymeleaf模板就是增加一些额外元素属性的HTML，这些属性能够指导模板如何渲染request数据。举例来说，如果有一个请求属性的key为“message”，我们想要使用Thymeleaf将其渲染到一个HTML <p>标签中，那么在Thymeleaf模板中我们可以这样写：

```
<p th:text="${message}">placeholder message</p>
```

当模板渲染成HTML的时候，<p>元素体将会被替换为Servlet Request中key为“message”的属性值。“th:text”是Thymeleaf命名空间中的属性，它会执行这个替换过程。${}会告诉它要使用某个请求属性（在本例中，也就是“message”）中的值。

Thymeleaf还提供了一个属性“th:each”，它会迭代一个元素集合，为集合中的每个条目渲染HTML。在我们设计视图展现模型中的配料列表时，这就非常便利了。举例来说，如果只想渲染“wrap”配料的列表，我们可以使用如下的HTML片段：

```
<h3>Designate your wrap:</h3>
<div th:each="ingredient : ${wrap}">
  <input name="ingredients" type="checkbox" th:value="${ingredient.id}" />
  <span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
```

在这里，我们在<div>标签中使用th:each属性，这样的话就能针对wrap request属性所对应集合中的每个元素重复渲染<div>了。在每次迭代的时候，配料元素都会绑定到一个名为ingredient的Thymeleaf变量上。

在<div>元素中，有一个<input>复选框元素，还有一个为复选框提供标签的<span>元素。复选框使用Thymeleaf的th:value来为渲染出的<input>元素设置value属性，这里会将其设置为所找到的ingredient的id属性。<span>元素使用th:text将“INGREDIENT”占位符文本替换为ingredient的name属性。

当用实际的模型数据进行渲染的时候，其中一个<div>迭代的渲染结果可能会如下所示：

```
<div>
  <input name="ingredients" type="checkbox" value="FLTO" />
  <span>Flour Tortilla</span><br/>
</div>
```

最终，上述的Thymeleaf片段会成为一大段HTML表单的一部分，我们taco艺术家用户会通过这个表单来提交其美味的作品。完整的Thymeleaf模板会包括所有的配料类型，表单如程序清单2.3所示：

程序清单2.3　设计taco的完整页面

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
  </head>

  <body>
    <h1>Design your taco!</h1>
    <img th:src="@{/images/TacoCloud.png}"/>

    <form method="POST" th:object="${design}">
    <div class="grid">
      <div class="ingredient-group" id="wraps">
      <h3>Designate your wrap:</h3>
      <div th:each="ingredient : ${wrap}">
        <input name="ingredients" type="checkbox" th:value="${ingredient.i
d}"
     />
         <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="proteins">
      <h3>Pick your protein:</h3>
      <div th:each="ingredient : ${protein}">
```

```
        <input name="ingredients" type="checkbox" th:value="${ingredient.i
d}"
     />
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="cheeses">
      <h3>Choose your cheese:</h3>
      <div th:each="ingredient : ${cheese}">
        <input name="ingredients" type="checkbox" th:value="${ingredient.i
d}"
     />
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="veggies">
      <h3>Determine your veggies:</h3>
      <div th:each="ingredient : ${veggies}">
        <input name="ingredients" type="checkbox" th:value="${ingredient.i
d}"
     />
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>

      <div class="ingredient-group" id="sauces">
      <h3>Select your sauce:</h3>
      <div th:each="ingredient : ${sauce}">
        <input name="ingredients" type="checkbox" th:value="${ingredient.i
d}"
     />
        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
      </div>
      </div>
      </div>

      <div>

      <h3>Name your taco creation:</h3>
      <input type="text" th:field="*{name}"/>
      <br/>

      <button>Submit your taco</button>
      </div>
    </form>
```

```
  </body>
</html>
```

可以看到，我们会为各种类型的配料重复定义<div>片段。另外，我们还包含了一个Submit按钮，以及用户用来定义其作品名称的输入域。

值得注意的是，完整的模板包含了一个Taco Cloud的logo图片以及对样式表的<link>引用[2]。在这两个场景中，都使用了Thymeleaf的@{}操作符，用来生成一个相对上下文的路径，以便于引用我们需要的静态制件（artifact）。正如我们在第1章中所学到的，在Spring Boot应用中，静态内容要放到根路径的“/static”目录下。

我们的控制器和视图已经完成了，现在我们可以将应用启动起来，看一下我们的劳动成果。运行Spring Boot应用有很多种方式。在第1章中，我为你首先展示了如何将应用构建成一个可执行的JAR文件，然后通过java –jar命令来运行这个JAR。我还展示了如何直接通过mvn spring-boot:run构建命令来运行应用。

不管你采用哪种方式来启动Taco Cloud应用，在启动之后都可以通过http://localhost: 8080/design来进行访问。你将会看到如图2.2所示的一个页面。

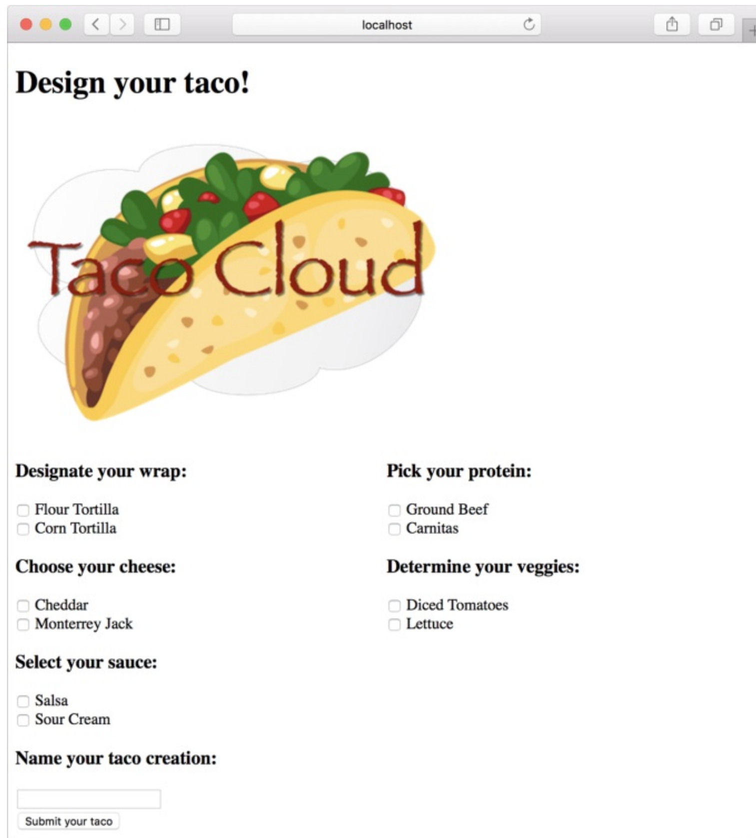


图2.2　渲染之后的taco设计页面

这看上去非常不错！访问你站点的taco艺术家可以看到一个表单，这个表单中包含了各种taco配料，他们可以使用这些配料创建自己的杰作。但是当他们点击Submit Your Taco按钮的时候会发生什么呢？

我们的DesignTacoController还没有为接收创建taco的请求做好准备。如果提交设计表单，用户就会遇到一个错误（具体来讲，将会是一个HTTP 405错误：Request Method “POST” Not Supported）。接下来，我们通过编写一些处理表单提交的控制器代码来修正这个错误。

2.2　处理表单提交

仔细看一下视图中的<form>标签，你将会发现它的method属性被设置成了POST。除此之外，<form>并没有声明action属性。这意味着当表单提交的时候，浏览器会收集表单中的所有数据，并以HTTP POST请求的形式将其发送至服务器端，发送路径与渲染表单的GET请求路径相同，也就是“/design”。

因此，在该POST请求的接收端，我们需要有一个控制器处理方法。在DesignTacoController中，我们会编写一个新的处理器方法来处理针对“/design”的POST请求。

在程序清单2.2中，我们曾经使用@GetMapping注解声明showDesignForm()方法要处理针对“/design”的HTTP GET请求。与@GetMapping处理GET请求类似，我们可以使用@PostMapping来处理POST请求。为了处理taco设计的表单提交，在DesignTacoController中添加程序清单2.4所述的processDesign()方法。

程序清单2.4　使用@PostMapping来处理POST请求

```
@PostMapping
public String processDesign(Taco design) {
  // Save the taco design...
  // We'll do this in chapter 3
  log.info("Processing design: " + design);

  return "redirect:/orders/current";
}
```

如processDesign()方法所示，@PostMapping与类级别的@RequestMapping协作，指定processDesign()方法要处理针对“/design”的POST请求。我们所需要的正是以这种方式处理taco艺术家的表单提交。

当表单提交的时候，表单中的输入域会绑定到Taco对象（这个类会在下面的程序清单中进行介绍）的属性中，该对象会以参数的形式传递给processDesign()。从这里开始，processDesign()就可以针对Taco对象采取任意操作了。

程序清单2.5　定义taco设计的领域对象

```
package tacos;
import java.util.List;
import lombok.Data;
@Data
public class Taco {

  private String name;
  private List<String> ingredients;

}
```

我们可以看到，Taco是一个非常简单的Java领域对象，其中包含了几项属性。与Ingredient类似，Taco类也添加了@Data注解，会在编译期自动生成必要的JavaBean方法，所以这些方法在运行期是可用的。

回过头来再看一下程序清单 2.3 中的表单，你会发现其中包含多个checkbox元素，它们的名字都是ingredients，另外还有一个名为name的文本输入元素。表单中的这些输入域直接对应Taco类的ingredients和name属性。

表单中的name输入域只需要捕获一个简单的文本值。因此，Taco的name属性是String类型的。配料的复选框也有文本值，但是用户可能会选择一个或多个，所以它们所绑定的ingredients属性是一个List<String>，能够捕获选中的每种配料。

processDesign()方法对Taco对象没有执行任何操作。实际上，这个方法什么都没做。现在，这样是可以的。到第3章，我们将会添加一些持久化的逻辑，将提交的Taco保存到一个数据库中。

与showDesignForm()方法类似，processDesign()最后也返回了一个String类型的值。同样与showDesignForm()相似，返回的这个值代表了一个要展现给用户的视图。但是，区别在于processDesign()返回的值带有“redirect:”前缀，表明这是一个重定向视图。更具体地讲，它表明在processDesign()完成之后，用户的浏览器将会重定向到相对路径“/order/current”。

这里的想法是在创建完taco后，用户将会被重定向到一个订单表单页面，在这里用户可以创建一个订单，将他们所创建的taco快递过去。但是，我们现在还没有处理“/orders/current”请求的控制器。

根据已经学到的关于@Controller、@RequestMapping和@GetMapping的知识，我们可以很容易地创建这样的控制器。它应该如程序清单2.6所示。

程序清单2.6　展现taco订单表单的控制器

```
package tacos.web;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import lombok.extern.slf4j.Slf4j;
import tacos.Order;
@Slf4j
```

```
@Controller
@RequestMapping("/orders")
public class OrderController {

  @GetMapping("/current")
  public String orderForm(Model model) {
    model.addAttribute("order", new Order());
    return "orderForm";
  }

}
```

在这里，我们再次使用Lombok @Slf4j注解在运行期创建一个SLF4J Logger对象。稍后，我们将会使用这个Logger记录所提交订单的详细信息。

类级别的@RequestMapping指明这个控制器的请求处理方法都会处理路径以“/orders”开头的请求。当与方法级别的@GetMapping注解结合之后，它就能够指定orderForm()方法，会处理针对“/orders/current”的HTTP GET请求。

orderForm()方法本身非常简单，只是返回了一个名为orderForm的逻辑视图名。在第3章学习完如何将所创建的taco保存到数据库之后，我们将会重新回到这个方法并对其进行修改，用一个Taco对象的列表来填充模型并将其放到订单中。

orderForm视图是由名为orderForm.html的Thymeleaf模板来提供的，如程序清单2.7所示。

程序清单2.7　一个taco订单表单视图

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
    xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Taco Cloud</title>
  <link rel="stylesheet" th:href="@{/styles.css}" />
</head>

<body>

  <form method="POST" th:action="@{/orders}" th:object="${order}">
    <h1>Order your taco creations!</h1>

    <img th:src="@{/images/TacoCloud.png}"/>
    <a th:href="@{/design}" id="another">Design another taco</a><br/>

    <div th:if="${#fields.hasErrors()}">
      <span class="validationError">
      Please correct the problems below and resubmit.
      </span>
    </div>
    <h3>Deliver my taco masterpieces to...</h3>
    <label for="name">Name: </label>
    <input type="text" th:field="*{name}"/>
    <br/>

    <label for="street">Street address: </label>
    <input type="text" th:field="*{street}"/>
    <br/>

    <label for="city">City: </label>
    <input type="text" th:field="*{city}"/>
    <br/>

    <label for="state">State: </label>
    <input type="text" th:field="*{state}"/>
    <br/>

    <label for="zip">Zip code: </label>
    <input type="text" th:field="*{zip}"/>
    <br/>

    <h3>Here's how I'll pay...</h3>
    <label for="ccNumber">Credit Card #: </label>
    <input type="text" th:field="*{ccNumber}"/>
    <br/>

    <label for="ccExpiration">Expiration: </label>
    <input type="text" th:field="*{ccExpiration}"/>
    <br/>
```

```
      <label for="ccCVV">CVV: </label>
      <input type="text" th:field="*{ccCVV}"/>
      <br/>

      <input type="submit" value="Submit order"/>
    </form>

  </body>
</html>
```

从很大程度上来讲，orderForm.html就是典型的HTML/Thymeleaf内容，不需要过多关注。但是，需要注意一点，这里的<form>标签和程序清单2.3中的<form>标签有所不同，它指定了一个表单的action。如果不指定action，那么表单将会以HTTP POST的形式提交到与展现该表单相同的URL上。在这里，我们明确指明表单要POST提交到“/orders”上（使用Thymeleaf的@{...}操作符指定相对上下文的路径）。

因此，我们需要在OrderController中添加另外一个方法，以便于处理针对“/orders”的POST请求。我们在第3章才会对订单进行持久化，在此之前，我们让它尽可能简单，如程序清单2.8所示。

程序清单2.8　处理taco订单的提交

```
@PostMapping
public String processOrder(Order order) {
  log.info("Order submitted: " + order);
  return "redirect:/";
}
```

当调用processOrder()方法处理所提交的订单时，我们会得到一个Order对象，它的属性绑定了所提交的表单域。Order与Taco非常相似，是一个非常简单的类，其中包含了订单的信息，如程序清单2.9所示。

程序清单2.9　taco订单的领域对象

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
public class Order {

  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String ccNumber;
  private String ccExpiration;
  private String ccCVV;
}
```

现在，我们已经开发了OrderController和订单表单的视图，接下来我们可以尝试运行一下。打开浏览器并访问http://localhost:8080/design，为taco选择一些配料，并点击“Submit Your Taco”按钮，将会看到如图2.3所示的一个表单。

图2.3　taco订单表单

填充表单的一些输入域并点击“Submit order”按钮。请关注应用的日志来查看你的订单信息。在我尝试运行的时候，日志条目如下所示（为了适应页面的宽度，重新进行了格式化）：

```
Order submitted: Order(name=Craig Walls,street1=1234 7th Street,
     city=Somewhere, state=Who knows?, zip=zipzap, ccNumber=Who can guess?
,
ccExpiration=Some day, ccCVV=See-vee-vee)
```

如果仔细查看上述测试订单的日志，就会发现尽管processOrder()方法完成了它的工作并处理了表单提交，但是它让一些错误的信息混入了进来。表单中的大多数输入域包含的可能都是不正确的信息。我们接下来添加一些校验，确保所提交的数据至少与所需的信息比较接近。

2.3 校验表单输入

在设计新的taco作品的时候，如果用户没有选择配料或者没有为他们的作品指定名称，那么将会怎样呢？当提交表单的时候，没有填写所需的地址输入域又将发生什么呢？或者，在信用卡域中输入了一个根本不合法的数字，又该怎么办呢？

就目前的情况来看，没有什么能够阻止用户在创建taco的时候不选择任何配料，或者输入空的快递地址，甚至将他们喜欢的歌词作为信用卡号进行提交。这是因为我们还没有指明这些输入域该如何进行校验。

有种校验方法就是在processDesign()和processOrder()方法中添加大量乱七八糟的if/then代码块，逐个检查，确保每个输入域都满足对应的校验规则。但是，这样会非常烦琐，并且难以阅读和调试。

比较幸运的是，Spring支持Java的Bean校验API（Bean Validation API，也被称为JSR-303）。这样的话，我们能够更容易地声明检验规则，而不必在应用程序代码中显式编写声明逻辑。借助Spring Boot，要在项目中添加校验库，我们甚至不需要做任何特殊的操作，这是因为Validation API以及Validation API 的Hibernate实现将会作为Spring Boot

web starter的传递性依赖自动添加到项目中。

要在Spring MVC中应用校验，我们需要。

- 在要被校验的类上声明校验规则：具体到我们的场景中，也就是Taco类。
- 在控制器方法中声明要进行校验：具体来讲，也就是DesignTacoController的processDesign()方法和OrderController的processOrder()方法。
- 修改表单视图以展现校验错误。

Validation API提供了一些可以添加到领域对象上的注解，以便于声明校验规则。Hibernate的Validation AP实现又添加了一些校验注解。接下来，我们看一下如何使用其中的一些注解来校验用户提交的Taco和Order。

2.3.1　声明校验规则

对于Taco类来说，我们想要确保name属性不能为空或null，同时希望选中的配料至少要包含一项。程序清单2.10将展示更新后的Taco类，它使用@NotNull和@Size注解来声明这些校验规则。

程序清单2.10　为Taco领域类添加校验

```
package tacos;
import java.util.List;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import lombok.Data;
```

```
@Data
public class Taco {

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<String> ingredients;

}
```

我们可以发现，除了要求name属性不为null之外，我们还声明了它的值在长度上至少要有5个字符。

在对提交的taco订单进行校验时，我们必须要给Order类添加注解。对于地址相关的属性，我们只想确保用户没有提交空白字段。为此，我们可以使用Hibernate Validator的@NotBlank注解。

但是，支付相关的字段就比较复杂了。我们不仅要确保ccNumber属性不为空，还要保证它所包含的值是一个合法的信用卡号码。ccExpiration属性必须符合MM/YY格式（两位的月份和年份）。ccCVV属性需要是一个3位的数字。为了实现这种校验，我们需要其他的一些Java Bean Validation API注解，并结合来自Hibernate Validator的注解。程序清单2.11展现了校验Order类所需的变更。

程序清单2.11　校验订单的字段

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import javax.validation.constraints.NotBlank;
import lombok.Data;

@Data
```

```
public class Order {

  @NotBlank(message="Name is required")
  private String name;

  @NotBlank(message="Street is required")
  private String street;

  @NotBlank(message="City is required")
  private String city;

  @NotBlank(message="State is required")
  private String state;

  @NotBlank(message="Zip code is required")
  private String zip;

  @CreditCardNumber(message="Not a valid credit card number")
  private String ccNumber;

  @Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$",
           message="Must be formatted MM/YY")
  private String ccExpiration;

  @Digits(integer=3, fraction=0, message="Invalid CVV")
  private String ccCVV;

}
```

我们可以看到，ccNumber属性添加了@CreditCardNumber注解。这个注解声明该属性的值必须是合法的信用卡号，它要能通过Luhn算法的检查。这能防止用户有意或无意地输入错误的数据，但是该检查并不能确保这个信用卡号真的分配给了某个账户，也不能保证这个账号能够用来进行支付。

令人遗憾的是，目前还没有现成的注解来校验ccExpiration 属性的MM/YY格式。在这里，我使用了@Pattern注解并为其提供了一个正则表达式，确保属性值符合预期的格式。如果你想知道如何解释这个正则

表达式，那么我建议你参考一些在线的正则表达式指南。正则表达式是一种魔法，已经超出了本书的范围。

最后，在ccCVV属性上添加了@Digits注解，能够确保它的值包含3位数字。

所有的校验注解都包含了一个message属性，该属性定义了当输入的信息不满足声明的校验规则时要给用户展现的消息。

2.3.2 在表单绑定的时候执行校验

现在，我们已经声明了如何校验Taco和Order，接下来我们要重新修改每个控制器，让表单在POST提交至对应的控制器方法时执行对应的校验。

要校验提交的Taco，我们需要为DesignTacoController中processDesign()方法的Taco参数添加一个Java Bean Validation API的@Valid注解，如程序清单2.12所示。

程序清单2.12　校验POST提交的Taco

```
@PostMapping
public String processDesign(@Valid Taco design, Errors errors) {
  if (errors.hasErrors()) {
    return "design";
  }

  // Save the taco design...
  // We'll do this in chapter 3
  log.info("Processing design: " + design);
  return "redirect:/orders/current";
```

```
}
```

@Valid注解会告诉Spring MVC要对提交的Taco对象进行校验，而校验时机是在它绑定完表单数据之后、调用processDesign()之前。如果存在校验错误，那么这些错误的细节将会捕获到一个Errors对象中并传递给processDesign()。processDesign()方法的前几行会查阅Errors对象，调用其hasErrors()方法判断是否有校验错误。如果存在校验错误，那么这个方法将不会处理Taco对象并返回“design”视图名，表单会重新展现。

为了对提交的Order对象进行校验，OrderController的processOrder()方法也需要进行类似的变更，如程序清单2.13所示。

程序清单2.13　校验POST提交的Order

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors) {
  if (errors.hasErrors()) {
    return "orderForm";
  }

  log.info("Order submitted: " + order);
  return "redirect:/";
}
```

在这两个场景中，如果没有校验错误，那么方法都会允许处理提交的数据。如果存在校验错误，那么请求将会被转发至表单视图上，以便让用户有机会纠正他们的错误。

但是，用户该如何知道有哪些要纠正的错误呢？如果我们无法指出表单上的错误，那么用户只能不断猜测如何才能成功提交表单。

2.3.3　展现校验错误

Thymeleaf提供了便捷访问Errors对象的方法，这就是借助fields及其th:errors属性。举例来说，为了展现信用卡字段的校验错误，我们可以添加一个<span>元素，该元素会将对错误的引用用到订单的表单模板上，如程序清单2.14所示。

程序清单2.14　展现校验错误

```
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}"/>
<span class="validationError"
      th:if="${#fields.hasErrors('ccNumber')}"
      th:errors="*{ccNumber}">CC Num Error</span>
```

在这里，<span>元素使用class属性来为错误添加样式，以引起用户的注意。除此之外，它还使用th:if属性来决定是否要显示该元素。fields属性的hasErrors()方法会检查ccNumber域是否存在错误，如果存在，就将会渲染<span>。

th:errors属性引用了ccNumber输入域，如果该输入域存在错误，那么它会将<span>元素的占位符内容替换为校验信息。

在为订单表单的其他输入域都添加类似的<span>标签之后，如果提交错误信息，那么表单将会如图2.4所示。错误信息提示姓名、城市和邮政编码字段为空，而且所有支付相关的输入域均未满足校验条件。

图2.4　在订单表单上展现校验错误

现在，我们的Taco Cloud控制器不仅能够展现和捕获输入，还能校验用户提交的信息是否满足一定的基本验证规则。接下来，我们后退一步，重新考虑一下第1章中的HomeController，介绍一种替代实现方案。

2.4　使用视图控制器

到目前为止，我们已经为Taco Cloud应用编写了3个控制器。尽管这

3个控制器服务于应用程序的不同功能，但是它们基本上都遵循相同的编程模型：

- 它们都使用了@Controller注解，表明它们是控制器类，并且应该被Spring的组件扫描功能自动发现并初始化为Spring应用上下文中的bean；
- 除了HomeController之外，其他的控制器都在类级别使用了@RequestMapping注解，据此定义该控制器所处理的基本请求模式；
- 它们都有一个或多个带@GetMapping或@PostMapping注解的方法，指明了该由哪个方法来处理某种类型的请求。

我们所编写的大部分控制器都将遵循这个模式。但是，如果一个控制器非常简单，不需要填充模型或处理输入（在我们的场景中，也就是HomeController），那么还有一种方式可以定义控制器。请参考下面的程序清单2.15来学习如何声明视图控制器：也就是只将请求转发到视图而不做其他事情的控制器。

程序清单2.15　声明视图控制器

```
package tacos.web;

import org.springframework.context.annotation.Configuration;
import
     org.springframework.web.servlet.config.annotation.ViewControllerRegis
try
     ;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
```

```
    registry.addViewController("/").setViewName("home");
  }

}
```

关于WebConfig，最需要注意的事情就是它实现了WebMvcConfigurer接口。WebMvcConfigurer定义了多个方法来配置Spring MVC。尽管只是一个接口，但是它提供了所有方法的默认实现，只需要覆盖所需的方法即可。在本例中，我们覆盖了addViewControllers方法。

addViewControllers()方法会接收一个ViewControllerRegistry对象，我们可以使用它注册一个或多个视图控制器。在这里，我们调用registry的addViewController()方法，将“/”传递了进去，视图控制器将会针对该路径执行GET请求。这个方法会返回ViewControllerRegistration对象，我们马上基于该对象调用了setViewName()方法，用它指明当请求“/”的时候要转发到“home”视图上。

如前文所示，我们用配置类中的几行代码就替换了HomeController类。现在，我们可以删除HomeController了，应用的功能应该和之前完全一样。唯一需要注意的是，我们要重新找到第1章中的HomeControllerTest类，从@WebMvcTest注解中移除对HomeController的引用，这样测试类的编译才不会报错。

在这里，我们创建了一个新的WebConfig配置类来存放视图控制器的声明。但是，所有的配置类都可以实现WebMvcConfigurer接口并覆盖addViewController方法。举例来说，我们可以将相同的视图控制器声明

添加到TacoCloudApplication引导类中，如下所示：

```
@SpringBootApplication
public class TacoCloudApplication implements WebMvcConfigurer {

  public static void main(String[] args) {
    SpringApplication.run(TacoCloudApplication.class, args);
  }

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
  }

}
```

采用扩展已有配置类的方式能够避免创建新的配置类，从而减少项目中制件的数量。但是，我倾向于为每种配置（Web、数据、安全等）创建新的配置类，这样能够保持应用的引导配置类尽可能地整洁和简单。

在视图控制器方面，或者更通俗地来讲，在控制器将请求所转发到的视图方面，到目前为止，我们都是使用Thymeleaf来实现所有视图的。我很喜欢Thymeleaf，但是你可能想要为你的应用选择不同的模板模型，下面让我们来看一下Spring所能支持的众多视图方案。

2.5 选择视图模板库

在大多数情况下，视图模板库的选择完全取决于个人喜好。Spring非常灵活，能够支持很多常见的模板方案。除了个别情况之外，你所选择的模板库本身甚至不知道它在与Spring协作[3]。

表2.2列出Spring Boot自动配置功能所支持的模板方案。

表**2.2**　支持的模板方案

模板	**Spring Boot starter**依赖
FreeMarker	spring-boot-starter-freemarker
Groovy Templates	spring-boot-starter-groovy-templates
Java Server Pages（JSP）	无（由Tomcat或Jetty提供）
Mustache	spring-boot-starter-mustache
Thymeleaf	spring-boot-starter-thymeleaf

通常来讲，你只需要选择想要的视图模板库，将其作为依赖项添加到构建文件中，然后就可以在“/templates”目录下（在基于Maven或Gradle构建的项目中，它会在“src/main/resources”目录下）编写模板了。Spring Boot会探测到你所选择的模板库，并自动配置为Spring MVC控制器生成视图所需的各种组件。

在Taco Cloud应用中，我们已经按照这种方式使用了Thymeleaf模板库。在第1章中，在初始化项目的时候，我们选择了Thymeleaf复选框。这样会自动将Spring Boot的Thymeleaf starter依赖添加到pom.xml文件

中。当应用启动的时候，Spring Boot的自动配置功能会探测到存在Thymeleaf并自动为我们配置Thymeleaf bean。我们所需要做的就是在“/templates”中开始编写模板。

如果你想要使用不同的模板库，只需要在项目初始化的时候选择它或者编辑已有的项目构建文件，将新选择的模板库添加进来即可。

例如，我们想要使用Mustache来替换Thymeleaf，没有问题！只需要找到pom.xml文件，并将如下的代码

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

替换为：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mustache</artifactId>
</dependency>
```

当然，我们还需要确保按照Mustache语法来编写模板，而不是再使用Thymeleaf标签。Mustache的特定用法（以及其他备选模板语言）超出了本书的范围，但是我在这里给你一个直观的印象，让你明白大致会是什么样子。如下代码是Mustache模板的一个片段，它能够渲染taco设计表单中的某个配料组：

```
<h3>Designate your wrap:</h3>
{{#wrap}}
<div>
  <input name="ingredients" type="checkbox" value="{{id}}" />
```

```
  <span>{{name}}</span><br/>
</div>
{{/wrap}}
```

这是2.1.3小节中Thymeleaf代码片段的Mustache等价实现。{{#wrap}}代码块（结尾对应使用{{/wrap}}）会遍历请求中key为wrap的属性并为每个条目渲染嵌入式HTML。{{id}}和{{name}}标签分别会引用每个条目（应该是一个Ingredient）的id和name属性。

你可能已经注意到了，在表2.2中，JSP并不需要在构建文件中添加任何特殊的依赖。这是因为Servlet容器本身（默认是Tomcat）会实现JSP，因此不需要额外的依赖。但是，如果你选择使用JSP，会有另外一个问题。事实上，Java Servlet容器包括嵌入式的Tomcat和Jetty容器，通常会在“/WEB-INF”目录下寻找JSP。如果我们将应用构建成一个可执行的JAR文件，就无法满足这种需求了。因此，只有在将应用构建为WAR文件并部署到传统的Servlet容器中时，才能选择JSP方案。如果你想要构建可执行的JAR文件，那么必须选择Thymeleaf、FreeMarker或表2.2中的其他方案。

缓存模板

默认情况下，模板只有在第一次使用的时候解析一次，解析的结果会被后续的请求所使用。对于生产环境来说，这是一个很棒的特性，它能防止每次请求时多余的模板解析过程，因此有助于提升性能。

但是，在开发期，这个特性就不太友好了。假设我们启动完应用之

后访问taco的设计页面，然后决定对它做一些修改，但是当我们刷新Web浏览器的时候显示的依然是原始的版本。要想看到变更效果，就必须重新启动应用，这当然是非常不方便的。

幸运的是，有一种方法可以禁用缓存。我们所需要做的就是将相关的缓存属性设置为false。表2.3列出每种模板库所对应的缓存属性。

表**2.3**　启用/禁用模板缓存的属性

模板	启用缓存的属性
FreeMarker	spring.freemarker.cache
Groovy Templates	spring.groovy.template.cache
Mustache	spring.mustache.cache
Thymeleaf	spring.thymeleaf.cache

默认情况下，这些属性都设置成了true，以便于启用缓存。我们可以将缓存属性设置为false，从而禁用所选模板引擎的缓存。例如，要禁用Thymeleaf缓存，我们只需要在application.properties中添加如下这行代码：

```
spring.thymeleaf.cache=false
```

唯一需要注意的是，在将应用部署到生产环境之前，一定要删除这一行代码（或者将其设置为true）。有一种方法是将该属性设置到profile中（我们将会在第5章讨论profile）。另外一种更简单的方式是使用Spring Boot的DevTools，就像我们在第1章中的做法一样。DevTools提供了很多非常有用的开发期特性，其中有一项功能就是禁用所有模板库的缓存，但是在应用部署的时候DevTools会将自身禁用掉（从而能够重新启用模板缓存）。

2.6　小结

- Spring提供了一个强大的Web框架，名为Spring MVC，能够用来为Spring应用开发Web前端。
- Spring MVC是基于注解的，通过像@RequestMapping、@GetMapping和@PostMapping这样的注解来启用请求处理方法的声明。
- 大多数的请求处理方法最终会返回一个视图的逻辑名称，比如Thymeleaf模板，请求会转发到这样的视图上（同时会带有任意的模型数据）。
- Spring MVC支持校验，这是通过Java Bean Validation API和Validation API的实现（如Hibernate Validator）完成的。
- 对于没有模型数据和逻辑处理的HTTP GET请求，可以使用视图控制器。
- 除了Thymeleaf之外，Spring支持各种视图方案，包括FreeMarker、Groovy Templates和Mustache。

---

[1]　如果你想更深入地了解应用领域，我推荐你阅读Eric Evans的《领

域驱动设计》（ISBN978-7-115-37675-6，人民邮电出版社出版）。

[2]　样式表的内容与我们的讨论无关，它只是包含了让配料两列显示的样式，避免出现一个很长的配料列表。

[3]　其中一个这样的例外情况就是Thymeleaf的Spring Security方言，我们将会在第4章进行讨论。

第3章　使用数据

本章内容：

- 使用Spring的JdbcTemplate
- 使用SimpleJdbcInsert插入数据
- 使用Spring Data声明JPA repository

大多数应用程序提供的不仅仅是一个漂亮的界面，虽然用户界面可能会提供一些与应用程序的交互，但是应用程序和静态Web站点的区别在于它所展现和存储的数据。

在Taco Cloud应用中，我们需要维护配料、taco和订单的信息。如果没有数据库来存储信息，那么这个应用在第2章的基础上也就没有什么进展了。

在本章中，我们将会为Taco Cloud应用添加对数据持久化的支持。首先，我们会使用Spring对JDBC（Java Database Connectivity）的支持来消除样板式代码。随后，我们会使用JPA（Java Persistence API）重写

数据repository，进一步消除更多的代码。

3.1　使用JDBC读取和写入数据

几十年以来，关系型数据库和SQL一直是数据持久化领域的首选方案。尽管近年来涌现了许多可选的数据库类型，但是关系型数据库依然是通用数据存储的首选，而且短期内不太可能撼动它的地位。

在处理关系型数据的时候，Java开发人员有多种可选方案，其中最常见的是JDBC和JPA。Spring同时支持这两种抽象形式，能够让JDBC或JPA的使用更加容易。在本节中，我们将会讨论Spring如何支持JDBC，然后会在3.2节讨论Spring对JPA的支持。

Spring对JDBC的支持要归功于JdbcTemplate类。JdbcTemplate提供了一种特殊的方式，通过这种方式，开发人员在对关系型数据库执行SQL操作的时候能够避免使用JDBC时常见的繁文缛节和样板式代码。

为了更好地理解JdbcTemplate的功能，我们首先看一个不使用JdbcTemplate的样例，看一下如何在Java中执行一个简单的查询，如程序清单3.1所示。

程序清单3.1　不使用JdbcTemplate查询数据库

```
@Override
public Ingredient findOne(String id) {
  Connection connection = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  try {
    connection = dataSource.getConnection();
```

```
    statement = connection.prepareStatement(
        "select id, name, type from Ingredient where id=?");
    statement.setString(1, id);
    resultSet = statement.executeQuery();
    Ingredient ingredient = null;
    if(resultSet.next()) {
      ingredient = new Ingredient(
          resultSet.getString("id"),
          resultSet.getString("name"),
          Ingredient.Type.valueOf(resultSet.getString("type")));
    }
    return ingredient;
  } catch (SQLException e) {
    // ??? What should be done here ???
  } finally {
    if (resultSet != null) {
      try {
        resultSet.close();
      } catch (SQLException e) {}
    }
    if (statement != null) {
      try {
        statement.close();
      } catch (SQLException e) {}
    }
    if (connection != null) {
      try {
        connection.close();
      } catch (SQLException e) {}
    }
  }
  return null;
}
```

我向你保证，在程序清单3.1中存在查询数据库获取配料的那几行代码，但我敢肯定你很难在JDBC代码段中将这个查询找出来，它被创建连接、创建语句以及关闭连接、语句和结果集的清理功能完全包围了起来。

在创建连接、创建语句或执行查询的时候，可能会出现很多错误。这就要求我们捕获SQLException，它对于找出哪里出现了问题或如何解

决问题可能有所帮助，也可能毫无用处。

SQLException是一个检查型异常，它需要在catch代码块中进行处理。但是，对于常见的问题，如创建到数据库的连接失败或者输入的查询有错误，在catch代码块中是无法解决的，并且有可能要继续抛出以便于上游进行处理。作为对比，我们看一下使用JdbcTemplate的方式，如程序清单3.2所示。

程序清单3.2　使用JdbcTemplate查询数据库

```
private JdbcTemplate jdbc;

@Override
public Ingredient findOne(String id) {
  return jdbc.queryForObject(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient, id);
}

private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
  return new Ingredient(
      rs.getString("id"),
      rs.getString("name"),
      Ingredient.Type.valueOf(rs.getString("type")));
}
```

程序清单3.2中的代码显然要比程序清单3.1中的原始JDBC示例简单得多。这里没有创建任何的连接和语句。而且，在方法完成之后不需要对这些对象进行清理。最后，这里也没有任何catch代码块中无法处理的异常。剩下的代码仅仅关注执行查询（调用JdbcTemplate的queryForObject()）和将结果映射到Ingredient对象（在mapRowToIngredient()方法中）上。

程序清单3.2中的代码仅仅是在Taco Cloud应用中使用JdbcTemplate持久化和读取数据的一个片段。接下来，我们着手实现让应用程序支持JDBC持久化的下一个步骤。我们首先要对领域对象进行一些调整。

3.1.1　调整领域对象以适应持久化

在将对象持久化到数据库的时候，通常最好有一个字段作为对象的唯一标识。Ingredient类现在已经有了一个id字段，但是我们还需要将id字段添加到Taco和Order类中。

除此之外，记录Taco和Order是何时创建的可能会非常有用。所以，我们还会为每个对象添加一个字段来捕获它所创建的日期和时间。程序清单3.3展现了Taco类中新增的id和createdAt字段。

程序清单3.3　为Taco类添加ID和时间戳字段

```
@Data
public class Taco {

  private Long id;

  private Date createdAt;

  ...

}
```

因为我们使用Lombok在运行时生成访问器方法，所以在这里只需要声明id和createdAt属性就可以了。在运行时，它们都会有对应的getter和setter方法。类似的变更还需要应用到Order类上，如下所示：

```
@Data
public class Order {

  private Long id;

  private Date placedAt;

  ...

}
```

同样，Lombok会自动生成访问器方法，所以这是Order类的唯一变更（如果因为某种原因你无法使用Lombok，就需要自行编写这些方法了）。

现在，我们的领域类已经为持久化做好了准备。接下来，我们看一下该如何使用JdbcTemplate实现数据库的读取和写入。

3.1.2 使用JdbcTemplate

在开始使用JdbcTemplate之前，我们需要将它添加到项目的类路径中。这一点非常容易，只需要将Spring Boot的JDBC starter依赖添加到构建文件中就可以了：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

我们还需要一个存储数据的数据库。对于开发来说，嵌入式的数据库就足够了。我比较喜欢H2嵌入式数据库，所以我会将如下的依赖添加到构建文件中：

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

随后，你将会看到如何配置应用以使用外部的数据库，现在我们看一下如何编写获取和保存Ingredient数据的repository。

定义JDBC repository

我们的Ingredient repository需要完成如下操作：

- 查询所有的配料信息，将它们放到一个Ingredient对象的集合中；
- 根据id，查询单个Ingredient；
- 保存Ingredient对象。

如下的IngredientRepository接口以方法声明的方式定义了3个操作：

```
package tacos.data;

import tacos.Ingredient;

public interface IngredientRepository {

  Iterable<Ingredient> findAll();

  Ingredient findOne(String id);

  Ingredient save(Ingredient ingredient);

}
```

尽管该接口敏锐捕捉到了配料repository都需要做些什么，但是我们依然需要编写一个IngredientRepository实现，使用JdbcTemplate来查询数

据库。程序清单3.4展示了编写该实现的第一步。

程序清单3.4　开始使用JdbcTemplate编写配料repository

```
package tacos.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;

@Repository
public class JdbcIngredientRepository
    implements IngredientRepository {

  private JdbcTemplate jdbc;

  @Autowired
  public JdbcIngredientRepository(JdbcTemplate jdbc) {
    this.jdbc = jdbc;
  }

  ...

}
```

我们可以看到，JdbcIngredientRepository添加了@Repository注解。Spring定义了一系列的构造型（stereotype）注解，@Repository是其中之一，其他注解还包括@Controller和@Component。为JdbcIngredientRepository添加@Repository注解之后，Spring的组件扫描就会自动发现它，并且会将其初始化为Spring应用上下文中的bean。

当Spring创建JdbcIngredientRepository bean的时候，它会通过@Autowired标注的构造器将JdbcTemplate注入进来。这个构造器将JdbcTemplate赋值给一个实例变量，这个变量会被其他方法用来执行数

据库查询和插入操作。说到其他的这些方法，让我们先看一下findAll()和findOne()的实现，如程序清单3.5所示。

程序清单3.5　使用JdbcTemplate查询数据库

```
@Override
public Iterable<Ingredient> findAll() {
  return jdbc.query("select id, name, type from Ingredient",
      this::mapRowToIngredient);
}

@Override
public Ingredient findOne(String id) {
  return jdbc.queryForObject(
      "select id, name, type from Ingredient where id=?",
      this::mapRowToIngredient, id);
}

private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
  return new Ingredient(
      rs.getString("id"),
      rs.getString("name"),
      Ingredient.Type.valueOf(rs.getString("type")));
}
```

findAll()和findOne()以相同的方式使用了JdbcTemplate。findAll()方法预期返回一个对象的集合，它使用了JdbcTemplate的query()方法。query()会接受要执行的SQL以及Spring RowMapper的一个实现（用来将结果集中的每行数据映射为一个对象）。query ()方法还能以最终参数的形式接收查询中所需的任意参数。但是，在本例中，我们不需要任何参数。

findOne()方法预期只会返回一个Ingredient对象，所以它使用了JdbcTemplate的queryForObject()方法，而不是query()方法。

queryForObject()方法的运行方式和query()非常类似，只不过它只返回一个对象，而不是对象的List。在本例中，它接受要执行的查询、RowMapper以及要获取的Ingredient的id，该id会替换查询中的“?”。

如程序清单3.5所示，findAll()和findOne()中的RowMapper参数都是通过对mapRowToIngredient()的方法引用指定的。在使用JdbcTemplate的时候，Java 8的方法引用和lambda表达式非常便利，它们能够替代显式的RowMapper实现。但是，如果因为某种原因，你想要或者必须使用显式RowMapper，那么如下的findOne()实现将阐述该如何按照这种方式进行编写。

```
@Override
public Ingredient findOne(String id) {
  return jdbc.queryForObject(
      "select id, name, type from Ingredient where id=?",
      new RowMapper<Ingredient>() {
        public Ingredient mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          return new Ingredient(
              rs.getString("id"),
              rs.getString("name"),
              Ingredient.Type.valueOf(rs.getString("type")));
        };
      }, id);
}
```

从数据库中读取数据只是问题的一部分。在有些情况下，我们必须先将数据写入数据库，这样才能进行读取。所以，我们接下来看一下该如何实现save()方法。

插入一行数据

JdbcTemplate的update()方法可以用来执行向数据库中写入或更新数据的查询语句。如程序清单3.6所示，它可以用来将数据插入到数据库中。

程序清单3.6　使用JdbcTemplate插入数据

```
@Override
public Ingredient save(Ingredient ingredient) {
  jdbc.update(
      "insert into Ingredient (id, name, type) values (?, ?, ?)",
      ingredient.getId(),
      ingredient.getName(),
      ingredient.getType().toString());
  return ingredient;
}
```

因为在这里不需要将ResultSet数据映射为对象，所以update()方法要比query()或queryForObject()简单得多。它只需要一个包含待执行SQL的String以及每个查询参数对应的值即可。在本例中，查询有3个参数，对应save()方法最后的3个参数，分别是配料的id、名称和类型。

JdbcIngredientRepository编写完成之后，我们就可以将其注入到DesignTacoController中了，然后使用它来提供Ingredient对象的列表，不用再使用硬编码的值（就像第2章中所做的那样）。修改后的DesignTacoController如程序清单3.7所示。

程序清单3.7　在控制器中注入和使用repository

```
@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

  private final IngredientRepository ingredientRepo;
```

```
  @Autowired
  public DesignTacoController(IngredientRepository ingredientRepo) {
    this.ingredientRepo = ingredientRepo;
  }

  @GetMapping
  public String showDesignForm(Model model) {
    List<Ingredient> ingredients = new ArrayList<>();
    ingredientRepo.findAll().forEach(i -> ingredients.add(i));

    Type[] types = Ingredient.Type.values();
    for (Type type : types) {
      model.addAttribute(type.toString().toLowerCase(),
          filterByType(ingredients, type));
    }

    return "design";
  }

  ...

}
```

需要注意的是，showDesignForm()方法的第二行调用了注入的IngredientRepository的findAll()方法。findAll()方法会从数据库中获取所有的配料，并将它们过滤成不同的类型，然后放到模型中。

现在，我们马上就能启动应用并尝试这些变更了。但是，在使用查询语句从Ingredient表中读取数据之前，我们需要先创建这个表并填充一些配料数据。

3.1.3 定义模式和预加载数据

除了Ingredient表之外，我们还需要其他的一些表来保存订单和设计信息。图 3.1描述了我们所需要的表以及这些表之间的关联关系。

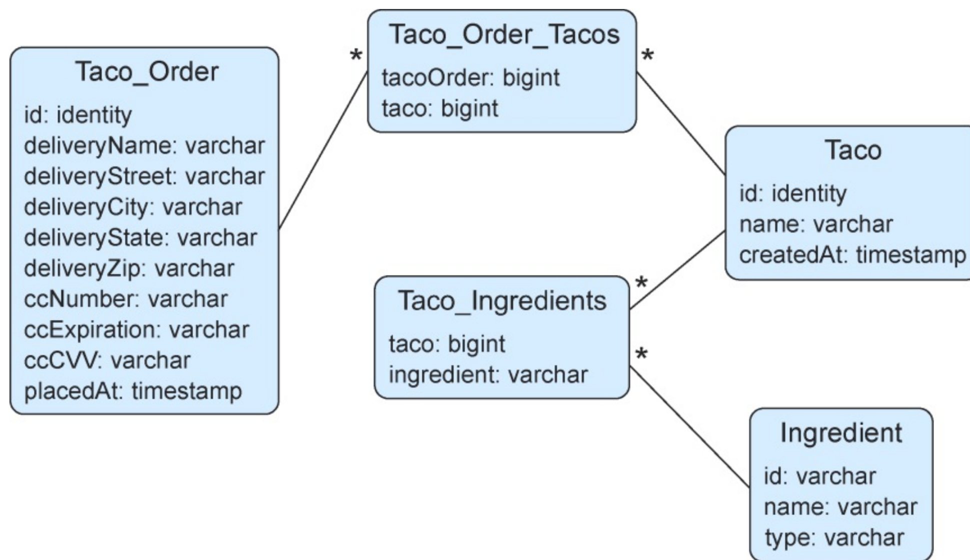


图3.1　Taco Cloud模式的表

图3.1中的表主要实现如下目的。

- Ingredient：保存配料信息。
- Taco：保存taco设计相关的信息。
- Taco_Ingredients：Taco中的每行数据都对应一行或多行，将taco和与之相关的配料映射在一起。
- Taco_Order：保存必要的订单细节。
- Taco_Order_Tacos：Taco_Order中的每行数据都对应一行或多行，将订单和与之相关的taco映射在一起。

程序清单3.8展示了创建表的SQL。

程序清单3.8　定义Taco Cloud的模式

```
create table if not exists Ingredient (
  id varchar(4) not null,
  name varchar(25) not null,
  type varchar(10) not null
);
```

```
create table if not exists Taco (
  id identity,
  name varchar(50) not null,
  createdAt timestamp not null
);

create table if not exists Taco_Ingredients (
  taco bigint not null,
  ingredient varchar(4) not null
);

alter table Taco_Ingredients
    add foreign key (taco) references Taco(id);
alter table Taco_Ingredients
    add foreign key (ingredient) references Ingredient(id);

create table if not exists Taco_Order (
  id identity,
    deliveryName varchar(50) not null,
    deliveryStreet varchar(50) not null,
    deliveryCity varchar(50) not null,
    deliveryState varchar(2) not null,
    deliveryZip varchar(10) not null,
    ccNumber varchar(16) not null,
    ccExpiration varchar(5) not null,
    ccCVV varchar(3) not null,
    placedAt timestamp not null
);

create table if not exists Taco_Order_Tacos (
  tacoOrder bigint not null,
  taco bigint not null
);

alter table Taco_Order_Tacos
    add foreign key (tacoOrder) references Taco_Order(id);
alter table Taco_Order_Tacos
    add foreign key (taco) references Taco(id);
```

现在，最大的问题是将这些模式定义放在什么地方。实际上，Spring Boot回答了这个问题。

如果在应用的根类路径下存在名为schema.sql的文件，那么在应用

启动的时候将会基于数据库执行这个文件中的SQL。所以，我们需要将程序清单3.8中的内容保存为名为schema.sql的文件并放到“src/main/resources”文件夹下。

我们可能还希望在数据库中预加载一些配料数据。幸运的是，Spring Boot还会在应用启动的时候执行根类路径下名为data.sql的文件。所以，我们可以使用程序清单3.9中的插入语句为数据库加载配料数据，并将其保存到“src/main/resources/data.sql”文件中。

程序清单3.9　预加载数据库

```
delete from Taco_Order_Tacos;
delete from Taco_Ingredients;
delete from Taco;
delete from Taco_Order;

delete from Ingredient;
insert into Ingredient (id, name, type)
                values ('FLTO', 'Flour Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
                values ('COTO', 'Corn Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
                values ('GRBF', 'Ground Beef', 'PROTEIN');
insert into Ingredient (id, name, type)
                values ('CARN', 'Carnitas', 'PROTEIN');
insert into Ingredient (id, name, type)
                values ('TMTO', 'Diced Tomatoes', 'VEGGIES');
insert into Ingredient (id, name, type)
                values ('LETC', 'Lettuce', 'VEGGIES');
insert into Ingredient (id, name, type)
                values ('CHED', 'Cheddar', 'CHEESE');
insert into Ingredient (id, name, type)
                values ('JACK', 'Monterrey Jack', 'CHEESE');
insert into Ingredient (id, name, type)
                values ('SLSA', 'Salsa', 'SAUCE');
insert into Ingredient (id, name, type)
                values ('SRCR', 'Sour Cream', 'SAUCE');
```

尽管我们目前只为配料数据编写了一个repository，但是你依然可以将Taco Cloud应用启动起来并访问设计页面，看一下JdbcIngredientRepository的实际功能。尽可以去尝试一下！当你尝试完回来的时候，我们将会编写Taco、Order和数据持久化的repository。

3.1.4 插入数据

我们已经粗略看到了如何使用JdbcTemplate将数据写入到数据库中。JdbcIngredient Repository的save()方法使用JdbcTemplate的update()方法将Ingredient对象保存到了数据库中。

尽管这是一个非常好的起步样例，但是它过于简单了。你马上将会看到保存数据可能会比JdbcIngredientRepository更加复杂。借助JdbcTemplate，我们有以下两种保存数据的方法。

- 直接使用update()方法。
- 使用SimpleJdbcInsert包装器类。

让我们首先看一下在持久化需求比保存Ingredient更为复杂的情况下该如何使用update()方法。

使用JdbcTemplate保存数据

现在，taco和order的repository唯一需要做的事情就是保存对应的对象。为了保存Taco对象，TacoRepository声明了一个save()方法：

```
package tacos.data;
```

```
import tacos.Taco;

public interface TacoRepository {

  Taco save(Taco design);

}
```

与之类似，OrderRepository也声明了一个save()方法：

```
package tacos.data;

import tacos.Order;

public interface OrderRepository {

  Order save(Order order);
}
```

看起来非常简单，对吧？但是，保存taco的时候需要同时将与该taco关联的配料保存到Taco_Ingredients表中。与之类似，保存订单的时候，需要同时将与该订单关联的taco保存到Taco_Order_Tacos表中。这样看来，保存taco和订单就会比保存配料更困难一些。

为了实现TacoRepository，我们需要用save()方法首先保存必要的taco设计细节（比如，名称和创建时间），然后对Taco对象中的每种配料都插入一行数据到Taco_Ingredients中。程序清单3.10展示了完整的JdbcTacoRepository类。

程序清单3.10　使用JdbcTemplate实现TacoRepository

```
package tacos.data;

import java.sql.Timestamp;
import java.sql.Types;
```

```java
import java.util.Arrays;
import java.util.Date;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;
import tacos.Taco;

@Repository
public class JdbcTacoRepository implements TacoRepository {

  private JdbcTemplate jdbc;

  public JdbcTacoRepository(JdbcTemplate jdbc) {
    this.jdbc = jdbc;
  }

  @Override
  public Taco save(Taco taco) {
    long tacoId = saveTacoInfo(taco);
    taco.setId(tacoId);
    for (Ingredient ingredient : taco.getIngredients()) {
      saveIngredientToTaco(ingredient, tacoId);
    }

    return taco;
  }

  private long saveTacoInfo(Taco taco) {
    taco.setCreatedAt(new Date());
    PreparedStatementCreator psc =
        new PreparedStatementCreatorFactory(
            "insert into Taco (name, createdAt) values (?, ?)",
            Types.VARCHAR, Types.TIMESTAMP
        ).newPreparedStatementCreator(
            Arrays.asList(
                taco.getName(),
                new Timestamp(taco.getCreatedAt().getTime())));

    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbc.update(psc, keyHolder);

    return keyHolder.getKey().longValue();
```

```
  }

  private void saveIngredientToTaco(
          Ingredient ingredient, long tacoId) {
    jdbc.update(
        "insert into Taco_Ingredients (taco, ingredient) " +
        "values (?, ?)",
        tacoId, ingredient.getId());
  }

}
```

我们可以看到，save()方法首先调用了私有的saveTacoInfo()方法，然后使用该方法所返回的taco ID来调用saveIngredientToTaco()，最后的这个方法会保存每种配料。这里的问题在于saveTacoInfo()方法的细节。

当向Taco中插入一行数据的时候，我们需要知道数据库生成的ID，这样我们才可以在每个配料信息中引用它。保存配料数据时所使用的update()方法无法帮助我们得到所生成的ID，所以在这里我们需要一个不同的update()方法。

这里的update()方法需要接受一个PreparedStatementCreator和一个KeyHolder。KeyHolder将会为我们提供生成的taco ID。但是，为了使用该方法，我们必须还要创建一个PreparedStatementCreator。

从程序清单3.10中可以看到，创建PreparedStatementCreator并不简单。首先需要创建PreparedStatementCreatorFactory，并将我们要执行的SQL传递给它，同时还要包含每个查询参数的类型。随后，需要调用该工厂类的newPreparedStatementCreator()方法，并将查询参数所需的值传递进来，这样才能生成一个PreparedStatementCreator。

有了PreparedStatementCreator之后，我们就可以调用update()方法了，并且需要将PreparedStatementCreator和KeyHolder（在本例中，也就是GeneratedKeyHolder的实例）传递进来。update()调用完成之后，我们就可以通过keyHolder.getKey().longValue()返回taco的ID。

回到save()方法，接下来我们会轮询Taco中的每个Ingredient，并调用saveIngredient ToTaco()。saveIngredientToTaco()使用更简单的update()形式来将对配料的引用保存到Taco_Ingredients表中。

对于TacoRepository来说，剩下的事情就是将它注入到DesignTacoController中，并在保存taco的时候调用它。程序清单3.11展现了注入repository所需的必要变更。

程序清单3.11　注入并使用TacoRepository

```
@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

  private final IngredientRepository ingredientRepo;

  private TacoRepository designRepo;

  @Autowired
  public DesignTacoController(
        IngredientRepository ingredientRepo,
        TacoRepository designRepo) {
    this.ingredientRepo = ingredientRepo;
    this.designRepo = designRepo;
  }

  ...

}
```

正如我们所看到的，构造器能够同时接受IngredientRepository和TacoRepository对象。该构造器将得到的对象赋值给实例变量，这样它们就可以在showDesignForm()和processDesign()中使用了。

谈到processDesign()方法，它的变更要比showDesignForm()的变更更大一些。程序清单3.12展现了新的processDesign()方法。

程序清单3.12　保存taco设计并将它们链接至订单页面

```
@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

  @ModelAttribute(name = "order")
  public Order order() {
    return new Order();
  }

  @ModelAttribute(name = "taco")
  public Taco taco() {
    return new Taco();
  }

  @PostMapping
  public String processDesign(
      @Valid Taco design, Errors errors,
      @ModelAttribute Order order) {

    if (errors.hasErrors()) {
      return "design";
    }

    Taco saved = designRepo.save(design);
    order.addDesign(saved);

    return "redirect:/orders/current";
  }

  ...
```

```
}
```

在程序清单3.12中，你首先关注到的事情可能是DesignTacoController类添加了@SessionAttributes("order")注解，并且它有一个新的带有@ModelAttribute注解的方法，即order()方法。与taco()方法类似，order()方法上的@ModelAttribute注解能够确保会在模型中创建一个Order对象。但是与模型中的Taco对象不同，我们需要订单信息在多个请求中都能出现，这样的话我们就能创建多个taco并将它们添加到该订单中。类级别的@SessionAttributes能够指定模型对象（如订单属性）要保存在session中，这样才能跨请求使用。

对taco设计的处理位于processDesign()方法中。该方法接受Order对象作为参数，同时还包括Taco和Errors对象。Order参数带有@ModelAttribute注解，表明它的值应该是来自模型的，Spring MVC不会尝试将请求参数绑定到它上面。

在检查完校验错误之后，processDesign()使用注入的TacoRepository来保存taco。然后，它将Taco对象保存到session里面的Order中。

实际上，在用户完成操作并提交订单表单之前，Order对象会一直保存在session中，并没有保存到数据库中。到时，OrderController需要调用OrderRepository的实现来保存订单。接下来，我们编写这个实现类。

使用**SimpleJdbcInsert**插入数据

在前文中提到，保存taco的时候不仅要将taco的名称和创建时间保存到Taco表中，还需要将该taco所引用的配料保存到Taco_Ingredients表中。此时，需要我们知道Taco的ID，而这是通过KeyHolder和PreparedStatementCreator获取的。

在保存订单的时候，存在类似的情况。我们不仅要将订单数据保存到Taco_Order表中，还要将订单对每个taco的引用保存到Taco_Order_Tacos表中。但是，在这里，我们不再使用烦琐的PreparedStatementCreator，而是引入SimpleJdbcInsert，这个对象对JdbcTemplate进行了包装，能够更容易地将数据插入到表中。

首先，我们要创建一个JdbcOrderRepository，它是OrderRepository的实现。但在编写save()方法的实现之前，我们先关注一下构造器，在构造器中我们会创建SimpleJdbcInsert的两个实例，分别用来把值插入到Taco_Order和Taco_Order_Tacos表中。程序清单3.13展现了JdbcOrderRepository（尚不包含save()方法）。

程序清单3.13　通过JdbcTemplate创建SimpleJdbcInsert

```
package tacos.data;

import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import com.fasterxml.jackson.databind.ObjectMapper;
```

```java
import tacos.Taco;
import tacos.Order;

@Repository
public class JdbcOrderRepository implements OrderRepository {

  private SimpleJdbcInsert orderInserter;
  private SimpleJdbcInsert orderTacoInserter;
  private ObjectMapper objectMapper;

  @Autowired
  public JdbcOrderRepository(JdbcTemplate jdbc) {
    this.orderInserter = new SimpleJdbcInsert(jdbc)
        .withTableName("Taco_Order")
        .usingGeneratedKeyColumns("id");

    this.orderTacoInserter = new SimpleJdbcInsert(jdbc)
        .withTableName("Taco_Order_Tacos");
    this.objectMapper = new ObjectMapper();
  }

...

}
```

与JdbcTacoRepository类似，JdbcOrderRepository通过构造器将JdbcTemplate注入进来。但是在这里，我们没有将JdbcTemplate直接赋给实例变量，而是使用它构建了两个SimpleJdbcInsert实例。第一个实例赋值给了orderInserter实例变量，配置为与Taco_Order表协作，并且假定id属性将会由数据库提供或生成。第二个实例赋值给了orderTacoInserter实例变量，配置为与Taco_Order_Tacos表协作，但是没有声明该表中ID是如何生成的。

该构造器还创建了Jackson中ObjectMapper类的一个实例，并将其赋值给一个实例变量。尽管Jackson的初衷是进行JSON处理，但是你很快就会看到我们是如何使用它来帮助我们保存订单和关联的taco的。

现在，我们看一下save()方法该如何使用SimpleJdbcInsert实例。程序清单3.14展示了save()方法以及一些私有方法，其中save()方法会将实际的工作委托给这些私有方法。

程序清单3.14　使用SimpleJdbcInsert插入数据

```
@Override
public Order save(Order order) {
  order.setPlacedAt(new Date());
  long orderId = saveOrderDetails(order);
  order.setId(orderId);
  List<Taco> tacos = order.getTacos();
  for (Taco taco : tacos) {
    saveTacoToOrder(taco, orderId);
  }

  return order;
}

private long saveOrderDetails(Order order) {
  @SuppressWarnings("unchecked")
  Map<String, Object> values =
      objectMapper.convertValue(order, Map.class);
  values.put("placedAt", order.getPlacedAt());

  long orderId =
      orderInserter
          .executeAndReturnKey(values)
          .longValue();
  return orderId;
}

private void saveTacoToOrder(Taco taco, long orderId) {
  Map<String, Object> values = new HashMap<>();
  values.put("tacoOrder", orderId);
  values.put("taco", taco.getId());
  orderTacoInserter.execute(values);
}
```

save()方法实际上没有保存任何内容，只是定义了保存Order及其关联的Taco对象的流程，并将实际的持久化任务委托给了

saveOrderDetails()和saveTacoToOrder()。

SimpleJdbcInsert有两个非常有用的方法来执行数据插入操作：execute()和execute AndReturnKey()。它们都接受Map<String, Object>作为参数，其中Map的key对应表中要插入数据的列名，而Map中的value对应要插入到列中的实际值。

我们只需将Order中的值复制到Map的条目中就能很容易地创建一个这样的Map。但是，Order有很多属性，这些属性与对应的列有着相同的名称。鉴于此，在saveOrderDetails()中，我决定使用Jackson的ObjectMapper及其convertValue()方法，以便于将Order转换为Map[1]。Map创建完成之后，我们将Map中placedAt条目的值设置为Order对象placedAt属性的值。之所以需要这样做，是因为ObjectMapper会将Date属性转换为long，这会导致与Taco_Order表中的placedAt字段不兼容。

当Map中准备好订单数据之后，我们就可以调用orderInserter的executeAnd ReturnKey()方法了。该方法会将订单信息保存到Taco_Order表中，并以Number对象的形式返回数据库生成的ID，继而调用longValue()方法将返回值转换为long类型。

saveTacoToOrder()方法要简单得多。在这里我们没有使用ObjectMapper将对象转换为Map，而是直接创建了一个Map并设置对应的值。同样，Map的key与表中的列名对应。我们只需要简单地调用orderTacoInserter的execute()方法就能执行插入操作了。

现在，我们可以将OrderRepository注入到OrderController中并开始使

用它了。程序清单3.15展示了完整的OrderController，包括使用注入的OrderRepository相关的变更。

程序清单3.15　在OrderController中使用OrderRepository

```
package tacos.web;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import tacos.Order;
import tacos.data.OrderRepository;

@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
public class OrderController {

  private OrderRepository orderRepo;

  public OrderController(OrderRepository orderRepo) {
    this.orderRepo = orderRepo;
  }

  @GetMapping("/current")
  public String orderForm() {
    return "orderForm";
  }

  @PostMapping
  public String processOrder(@Valid Order order, Errors errors,
                             SessionStatus sessionStatus) {
    if (errors.hasErrors()) {
      return "orderForm";
    }

    orderRepo.save(order);
    sessionStatus.setComplete();
```

```
    return "redirect:/";
  }

}
```

除了将OrderRepository注入到控制器中，OrderController唯一明显的变化就是processOrder()方法。在这个方法中，通过表单提交的Order对象（同时也是session中持有的Object对象）会通过注入的OrderRepository的save()方法进行保存。

订单保存完成之后，我们就不需要在session中持有它了。实际上，如果我们不把它清理掉，那么订单会继续保留在session中，其中包括与之关联的taco，下一次的订单将会从旧订单中保存的taco开始。所以，processOrder()方法请求了一个SessionStatus参数，并调用了它的setComplete()方法来重置session。

所有JDBC持久化代码已经就绪，现在我们可以启动Taco Cloud应用并进行尝试了。你可以按照自己的意愿创建任意数量的taco和订单。

你可能会发现，深入研究一下数据库中的内容是非常有帮助的。我们目前使用H2作为嵌入式数据库并且启用了Spring Boot DevTools，所以我们可以在浏览器中访问http://localhost:8080/h2-console以查看H2 Console。使用默认的凭证应该就可以进入，但是你需要确保JDBC URL字段设置成了dbc:h2:mem:testdb。登录之后，我们可以对Taco Cloud模式下的表执行任意的查询。

相对于普通的JDBC，Spring的JdbcTemplate和SimpleJdbcInsert能够极大地简化关系型数据库的使用。但是，你会发现使用JPA会更加简

单。我们回顾一下自己的工作内容，看一下Spring Data是如何让数据持久化变得更简单的。

3.2 使用Spring Data JPA持久化数据

Spring Data是一个非常大的伞形项目，由多个子项目组成，其中大多数子项目都关注对不同的数据库类型进行数据持久化。比较流行的几个Spring Data项目包括：

- Spring Data JPA：基于关系型数据库进行JPA持久化。
- Spring Data MongoDB：持久化到Mongo文档数据库。
- Spring Data Neo4j：持久化到Neo4j图数据库。
- Spring Data Redis：持久化到Redis key-value存储。
- Spring Data Cassandra：持久化到Cassandra数据库。

Spring Data为所有项目提供了一项最有趣且最有用的特性，就是基于repository规范接口自动生成repository的功能。

要了解Spring Data是如何运行的，我们需要重新开始，将本章前文基于JDBC的repository替换为使用Spring Data JPA的repository。首先，我们需要将Spring Data JPA添加到项目的构建文件中。

3.2.1 添加Spring Data JPA到项目中

Spring Boot应用可以通过JPA starter来添加Spring Data JPA。这个starter依赖不仅会引入Spring Data JPA，还会传递性地将Hibernate作为

JPA实现引入进来：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

如果你想要使用不同的JPA实现，那么至少需要将Hibernate依赖排除出去并将你所选择的JPA库包含进来。举例来说，如果想要使用EclipseLink来替代Hibernate，就需要像这样修改构建文件：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>hibernate-entitymanager</artifactId>
      <groupId>org.hibernate</groupId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.5.2</version>
</dependency>
```

需要注意，根据你所选择的JPA实现，这里可能还需要其他的变更。你可以参考所选择的JPA实现文档以了解更多细节。现在，我们重新看一下领域对象，并为它们添加注解，使其支持JPA持久化。

3.2.2　将领域对象标注为实体

你马上将会看到，在创建repository方面，Spring Data为我们做了很

多非常棒的事情。但是，在使用JPA映射注解标注领域对象方面，它却没有提供太多的助益。我们需要打开Ingredient、Taco和Order类，并为其添加一些注解，首先是Ingredient类，如程序清单3.16所示。

程序清单3.16　为Ingredient添加注解使其支持JPA持久化

```
package tacos;

import javax.persistence.Entity;
import javax.persistence.Id;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Entity
public class Ingredient {

  @Id
  private final String id;
  private final String name;
  private final Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

为了将Ingredient声明为JPA实体，它必须添加@Entity注解。它的id属性需要使用@Id注解，以便于将其指定为数据库中唯一标识该实体的属性。

除了JPA特定的注解，你可能会发现我们在类级别添加了

@NoArgsConstructor注解。JPA需要实体有一个无参的构造器，Lombok的@NoArgsConstructor注解能够帮助我们实现这一点。但是，我们不想直接使用它，因此通过将access属性设置为AccessLevel.PRIVATE使其变成私有的。因为这里有必须要设置的final属性，所以我们将force设置为true，这样Lombok生成的构造器就会将它们设置为null。

我们还添加了一个@RequiredArgsConstructor注解。@Data注解会为我们添加一个有参构造器，但是使用@NoArgsConstructor注解之后，这个构造器就会被移除掉。现在，我们显式添加@RequiredArgsConstructor注解，以确保除了private的无参构造器之外，我们还会有一个有参构造器。

接下来，我们看一下程序清单3.17所示的Taco类，看看它是如何标注为JPA实体的。

程序清单3.17　将Taco标注为实体

```
package tacos;
import java.util.Date;
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
@Entity
```

```
public class Taco {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  private Date createdAt;

  @ManyToMany(targetEntity=Ingredient.class)
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<Ingredient> ingredients;

  @PrePersist
  void createdAt() {
    this.createdAt = new Date();
  }
}
```

与Ingredient类似，Taco类现在添加了@Entity注解，并为其id属性添加了@Id注解。因为我们要依赖数据库自动生成ID值，所以在这里还为id属性设置了@GeneratedValue，将它的strategy设置为AUTO。

为了声明Taco与其关联的Ingredient列表之间的关系，我们为ingredients添加了@ManyToMany注解。每个Taco可以有多个Ingredient，而每个Ingredient可以是多个Taco的组成部分。

你会看到，在这里有一个新的方法createdAt()，并使用了@PrePersist注解。在Taco持久化之前，我们会使用这个方法将createdAt设置为当前的日期和时间。最后，我们要将Order对象标注为实体。程序清单3.18展示了新的Order类。

程序清单3.18　将Order标注为JPA实体

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
@Entity
@Table(name="Taco_Order")
public class Order implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  private Date placedAt;

  ...

  @ManyToMany(targetEntity=Taco.class)
  private List<Taco> tacos = new ArrayList<>();

  public void addDesign(Taco design) {
    this.tacos.add(design);
  }

  @PrePersist
  void placedAt() {
    this.placedAt = new Date();
  }
```

```
}
```

我们可以看到，Order所需的变更就是Taco的翻版。但是，在类级别这里有了一个新的注解，即@Table。它表明Order实体应该持久化到数据库中名为Taco_Order的表中。

我们可以将这个注解用到所有的实体上，但是只有Order有必要这样做。如果没有它，JPA默认会将实体持久化到名为Order的表中，但是order是SQL的保留字，这样做的话会产生问题。实体都已经标注好了，现在我们该编写repository了。

3.2.3 声明JPA repository

在JDBC版本的repository中，我们显式声明想要repository提供的方法。但是，借助 Spring Data，我们可以扩展CrudRepository接口。举例来说，如下是新的IngredientRepository接口。

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

CrudRepository定义了很多用于CRUD（创建、读取、更新、删除）操作的方法。注意，它是参数化的，第一个参数是repository要持久化的

实体类型，第二个参数是实体ID属性的类型。对于IngredientRepository来说，参数应该是Ingredient和String。

我们可以非常简单地定义TacoRepository：

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Taco;

public interface TacoRepository
         extends CrudRepository<Taco, Long> {

}
```

IngredientRepository和TacoRepository之间唯一比较明显的区别就是CrudRepository的参数。在这里，我们将其设置为Taco和Long，从而指定Taco实体（及其ID类型）是该repository接口的持久化单元。最后，相同的变更可以用到OrderRepository上：

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Order;

public interface OrderRepository
         extends CrudRepository<Order, Long> {

}
```

现在，我们有了3个repository。你可能会想，我们应该需要编写它们的实现类，包括每个实现类所需的十多个方法。但是，Spring Data JPA带来的好消息是，我们根本就不用编写实现类！当应用启动的时

候，Spring Data JPA会在运行期自动生成实现类。这意味着，我们现在就可以使用这些repository了。我们只需要像使用基于JDBC的实现那样将它们注入控制器中就可以了。

CrudRepository所提供的方法对于实体的通用持久化是非常有用的。但是，如果我们的需求并不局限于基本持久化，那又该怎么办呢？接下来，我们看一下如何自定义repository来执行特定领域的查询。

3.2.4 自定义JPA repository

假设除了CrudRepository提供的基本CRUD操作之外，我们还需要获取投递到指定邮编（Zip）的订单。实际上，我们只需要添加如下的方法声明到OrderRepository中，这个问题就解决了：

```
List<Order> findByDeliveryZip(String deliveryZip);
```

当创建repository实现的时候，Spring Data会检查repository接口的所有方法，解析方法的名称，并基于被持久化的对象来试图推测方法的目的。本质上，Spring Data定义了一组小型的领域特定语言（Domain-Specific Language，DSL），在这里持久化的细节都是通过repository方法的签名来描述的。

Spring Data能够知道这个方法是要查找Order的，因为我们使用Order对CrudRepository进行了参数化。方法名findByDeliveryZip()确定该方法需要根据deliveryZip属性相匹配来查找Order，而deliveryZip的值是作为参数传递到方法中来的。

findByDeliveryZip ()方法非常简单，但是Spring Data也能处理更加有意思的方法名称。repository方法是由一个动词、一个可选的主题（Subject）、关键词By以及一个断言所组成的。在findByDeliveryZip()这个样例中，动词是find，断言是DeliveryZip，主题并没有指定，暗含的主题是Order。

我们考虑另外一个更复杂的样例。假设我们想要查找投递到指定邮编且在一定时间范围内的订单。在这种情况下，我们可以将如下的方法添加到OrderRepository中，它就能达到我们的目的。

```
List<Order> readOrdersByDeliveryZipAndPlacedAtBetween(
      String deliveryZip, Date startDate, Date endDate);
```

图 3.2 展现了Spring Data在生成repository实现的时候是如何解析和理解readOrdersByDeliveryZipAndPlacedAtBetween()方法的。我们可以看到，在readOrdersByDeliveryZipAndPlacedAtBetween()中，动词是read。Spring Data会将get、read和find视为同义词，它们都是用来获取一个或多个实体的。另外，我们还可以使用count作为动词，它会返回一个int值，代表匹配实体的数量。



图3.2　Spring Data解析repository方法签名来确定要执行的查询

尽管方法的主题是可选的，但是这里要查找的就是Order。Spring Data会忽略主题中大部分的单词，所以你尽可以将方法命名为readPuppiesBy...，它依然会去查找Order实体，因为CrudRepository的类型是参数化的。

单词By后面的断言是方法签名中最为有意思的一部分。在本例中，断言指定了Order的两个属性：deliveryZip和placedAt。deliveryZip属性的值必须要等于方法第一个参数传入的值。关键字Between表明placedAt属性的值必须要位于方法最后两个参数的值之间。

除了Equals和Between操作之外，Spring Data方法签名还能包括如下的操作符：

- IsAfter、After、IsGreaterThan、GreaterThan
- IsGreaterThanEqual、GreaterThanEqual
- IsBefore、Before、IsLessThan、LessThan
- IsLessThanEqual、LessThanEqual
- IsBetween、Between
- IsNull、Null
- IsNotNull、NotNull
- IsIn、In
- IsNotIn、NotIn
- IsStartingWith、StartingWith、StartsWith
- IsEndingWith、EndingWith、EndsWith
- IsContaining、Containing、Contains
- IsLike、Like
- IsNotLike、NotLike

- IsTrue、True
- IsFalse、False
- Is、Equals
- IsNot、Not
- IgnoringCase、IgnoresCase

作为IgnoringCase/IgnoresCase的替代方案，我们还可以在方法上添加AllIgnoringCase或AllIgnoresCase，这样它就会忽略所有String对比的大小写。例如，请看如下方法：

```
List<Order> findByDeliveryToAndDeliveryCityAllIgnoresCase(
        String deliveryTo, String deliveryCity);
```

最后，我们还可以在方法名称的结尾处添加OrderBy，实现结果集根据某个列排序。例如，我们可以按照deliveryTo属性排序：

```
List<Order> findByDeliveryCityOrderByDeliveryTo(String city);
```

尽管方法名称约定对于相对简单的查询非常有用，但是，不难想象，对于更为复杂的查询，方法名可能会面临失控的风险。在这种情况下，可以将方法定义为任何你想要的名称，并为其添加@Query注解，从而明确指明方法调用时要执行的查询，如下面的样例所示：

```
@Query("Order o where o.deliveryCity='Seattle'")
List<Order> readOrdersDeliveredInSeattle();
```

在本例中，通过使用@Query，我们声明只查询所有投递到Seattle的订单。但是，我们可以使用@Query执行任何想要的查询，有些查询是通过方法命名约定很难甚至根本无法实现的。

3.3　小结

- Spring的JdbcTemplate能够极大地简化JDBC的使用。
- 在我们需要知道数据库所生成的ID值时，可以组合使用PreparedStatementCreator和KeyHolder。
- 为了简化数据的插入，可以使用SimpleJdbcInsert。
- Spring Data JPA能够极大地简化JPA持久化，我们只需编写repository接口即可。

---

[1]　我要承认这里对ObjectMapper的使用并不高明，但是我们毕竟已经将Jackson引入到了类路径中，这是由Spring Boot的web starter引入的。另外，使用ObjectMapper将对象映射为Map要比复制对象的每个属性到Map中容易得多。你尽可以使用其他技术来构建Inserter对象所需的Map，以替换对ObjectMapper的使用。

第4章　保护Spring

本章内容：

- 自动配置Spring Security
- 设置自定义的用户存储
- 自定义登录页
- 防范CSRF攻击
- 知道用户是谁

有一点不知道你是否在意过，那就是在电视剧中大多数人从不锁门。在*Leave it to Beaver*热播的时代，人们不锁门这事并不值得大惊小怪，但是在这个隐私和安全被看得极其重要的年代，看到电视剧中的角色允许别人大摇大摆地进入自己的寓所或家中实在让人难以置信。

现在，信息可能是我们最有价值的东西，一些不怀好意的人想尽办法试图偷偷进入不安全的应用程序来窃取我们的数据和身份信息。作为

软件开发人员，我们必须采取措施来保护应用程序中的信息。无论你是通过用户名/密码来保护电子邮件账号，还是基于交易PIN来保护经纪账户，安全性都是绝大多数应用系统中的一个重要切面。

4.1　启用Spring Security

保护Spring应用的第一步就是将Spring Boot security starter依赖添加到构建文件中。在项目的pom.xml文件中，添加如下的<dependency>条目：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

如果你使用Spring Tool Suite，那么这个过程会更加简单。用鼠标右键点击pom.xml文件并在Spring弹出菜单中选择“Edit Starters”，将会出现“Edit Spring Boot Starters”对话框，在“Core”分类中选择“Security”条目，如图4.1所示。

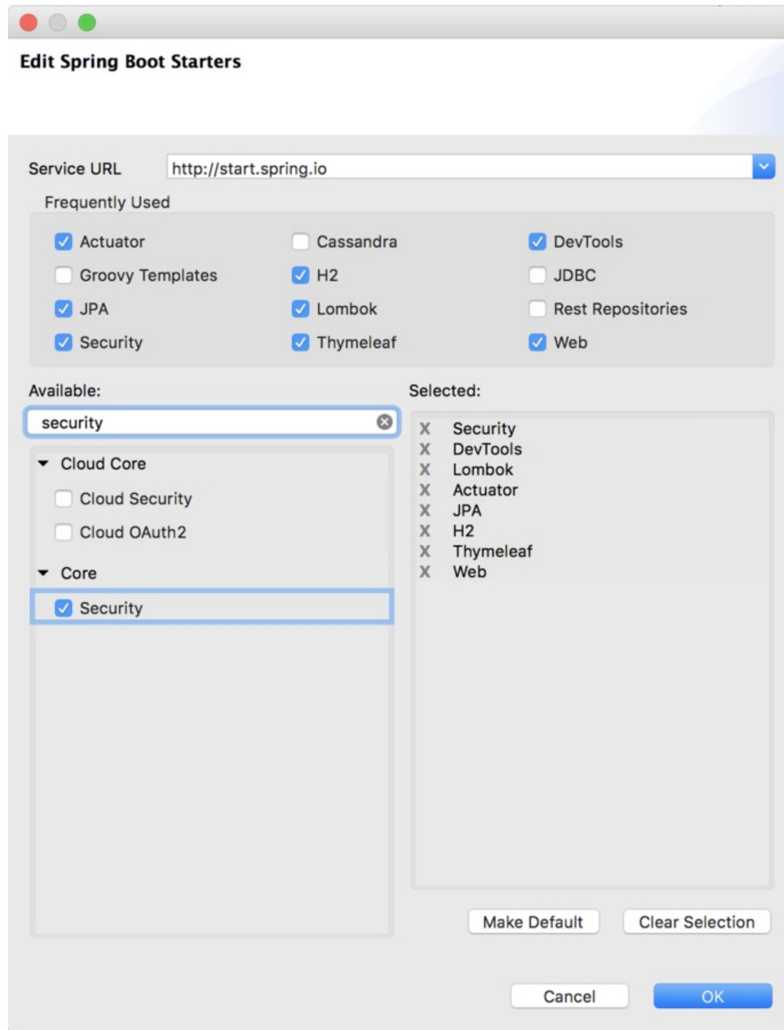


图4.1　使用Spring Tool Suite添加security starter

不管你是否相信，要保护我们的应用，只需添加这项依赖就可以了。当应用启动的时候，自动配置功能会探测到Spring Security出现在了类路径中，因此它会初始化一些基本的安全配置。

如果你想试一下，可以启动应用并尝试访问主页（或者任意页面）。应用将会弹出一个HTTP basic认证对话框并提示你进行认证。要想通过这个认证，你需要一个用户名和密码。用户名为user，而密码则是随机生成的，它会被写入应用的日志文件中。日志条目大致如下所

示：

```
Using default security password: 087cfc6a-027d-44bc-95d7-cbb3a798a1ea
```

假设输入了正确的用户名和密码，你就有权限访问应用了。

看上去保护Spring应用是一项非常简单的任务。现在，Taco Cloud应用已经安全了，我们可以结束本节并进入下一个话题了。但是，在继续下一步之前，我们回顾一下自动配置提供了什么类型的安全性。

通过将security starter添加到项目的构建文件中，我们得到了如下的安全特性：

- 所有的HTTP请求路径都需要认证；
- 不需要特定的角色和权限；
- 没有登录页面；
- 认证过程是通过HTTP basic认证对话框实现的；
- 系统只有一个用户，用户名为user。

这是一个很好的开端，但是我相信大多数应用（包括Taco Cloud）的安全需求与这些基础的安全特性截然不同。

如果想要确保Taco Cloud应用的安全性，我们还有很多的工作要做。我们至少要配置Spring Security实现如下功能：

- 通过登录页面来提示用户进行认证，而不是使用HTTP basic对话框；
- 提供多个用户，并提供一个注册页面，这样Taco Cloud的新用户能够注册进来；

- 对不同的请求路径，执行不同的安全规则。举例来说，主页和注册页面根本不需要进行认证。

为了满足Taco Cloud的安全需求，我们需要编写一些显式的配置，覆盖掉自动配置为我们提供的功能。我们首先配置一个合适的用户存储，这样就能有多个用户了。

4.2 配置Spring Security

多年以来，出现了多种配置Spring Security的方式，包括冗长的基于XML的配置。幸运的是，最近几个版本的Spring Security都支持基于Java的配置，这种方式更加易于阅读和编写。

在本章结束之前，我们会使用基于Java的Spring Security配置完成Taco Cloud安全性需要的所有设置。但是，首先，我们需要编写程序清单4.1中这个基础的配置类。

程序清单4.1　Spring Security的基础配置类

```
package tacos.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web
                        .configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
                    .configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

这个基础的安全配置都为我们做了些什么呢？其实并不太多，但是它确实朝着我们所需的安全性需求向前推进了一步。如果此时你再次访问Taco Cloud主页，那么系统依然会提示你进行登录。但是，现在不再是提示HTTP basic认证的对话框，而是会展现如图4.2所示的登录表单。

localhost

Login with Username and Password

User:

Password:

Login

图4.2　Spring Security为我们免费提供的简单登录页

提示：在进行手动安全测试的时候，你会发现将浏览器设置为私有或隐身模式会非常有用。这能够确保每次打开一个私有/隐身窗口都会有一个新的会话。每次你都需要重新登录应用，但是你尽可以放心，在安全性方面做得所有变更都会生效，旧会话不会有任何残留，妨碍我们看到变更的效果。

这是一个很小的改进，通过Web页面提示登录（尽管看上去非常简陋）要比HTTP basic对话框更友好一些。在4.3.2小节，我们将会自定义登录页面。不过，我们现在的任务是配置用户存储，使系统能够处理多个用户。

事实上，Spring Security为配置用户存储提供了多个可选方案，包括：

- 基于内存的用户存储；
- 基于JDBC的用户存储；
- 以LDAP作为后端的用户存储；
- 自定义用户详情服务。

不管使用哪种用户存储，你都可以通过覆盖WebSecurityConfigurerAdapter基础配置类中定义的configure()方法来进行配置。首先，我们可以将如下的方法添加到SecurityConfig类中：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  ...

}
```

现在，我们需要使用指定的AuthenticationManagerBuilder替换上面的省略号，以此来定义在认证过程中如何查找用户。我们先来看一下基于内存的用户存储。

4.2.1　基于内存的用户存储

用户信息可以存储在内存之中。假设我们只有数量有限的几个用户，而且这些用户几乎不会发生变化，在这种情况下，将这些用户定义成安全配置的一部分是非常简单的。

例如，程序清单4.2展示了如何在内存用户存储中配置两个用户，即“buzz”和“woody”。

程序清单4.2　在内存用户存储中定义用户

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .inMemoryAuthentication()
      .withUser("buzz")
        .password("infinity")
        .authorities("ROLE_USER")
      .and()
      .withUser("woody")
        .password("bullseye")
        .authorities("ROLE_USER");

}
```

我们可以看到，AuthenticationManagerBuilder使用构造者（builder）风格的接口来构建认证细节。在本例中，我们在安全配置中调用inMemoryAuthentication()方法来指定用户信息。

每次调用withUser()都会配置一个用户，这个方法给定的值是用户名，而密码和授权信息是通过password()和authorities()方法来指定的。如程序清单4.2中所示，两个用户都授予了ROLE_USER权限。用户buzz的密码为“infinity”，而woody的密码为“bullseye”。

对于测试和简单的应用来讲，基于内存的用户存储是很有用的，但是这种方式不能很方便地编辑用户。如果需要新增、移除或变更用户，那么你要对代码做出必要的修改，然后重新构建和部署应用。

对于Taco Cloud应用来说，我们希望顾客能够在应用中进行注册，并且能够管理自己的用户账号。这明显与内存用户存储的限制不符，所以我们接下来看一下另外一种方式，这种方式允许使用数据库后端作为用户存储。

4.2.2　基于JDBC的用户存储

用户信息通常会在关系型数据库中进行维护，基于JDBC的用户存储方案会更加合理一些。程序清单4.3展示了使用JDBC对存储在关系型数据库中的用户信息进行认证所需的Spring Security配置。

程序清单4.3　基于JDBC用户存储进行认证

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource);
}
```

在这里的configure()实现中，调用了AuthenticationManagerBuilder的jdbcAuthentication()方法。我们必须还要设置一个DataSource，这样它才

能知道如何访问数据库。这里的DataSource是通过自动装配的技巧获取到的。

重写默认的用户查询功能

尽管最少配置能够让一切运转起来，但是它对我们的数据库模式有一些要求，预期某些存储用户数据的表已经存在。更具体来说，下面的代码片段来源于Spring Security内部，并展现了当查找用户信息时所执行的SQL查询语句：

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
        "select username,password,enabled " +
        "from users " +
        "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
        "select username,authority " +
        "from authorities " +
        "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
        "select g.id, g.group_name, ga.authority " +
        "from groups g, group_members gm, group_authorities ga " +
        "where gm.username = ? " +
        "and g.id = ga.group_id " +
        "and g.id = gm.group_id";
```

在第一个查询中，我们获取了用户的用户名、密码以及是否启用的信息，用来进行用户认证。接下来的查询查找了用户所授予的权限，用来进行鉴权。在最后一个查询中，查找了用户作为群组的成员所授予的权限。

如果你能够在数据库中定义和填充满足这些查询的表，那么基本上就不需要再做什么额外的事情了。但是，也有可能你的数据库与上述的

不一致，那么你会希望在查询上有更多的控制权。如果是这样，那么我们可以按照程序清单4.4所示的方式配置自己的查询：

程序清单4.4　自定义用户详情查询

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?");

}
```

在本例中，我们只重写了认证和基本权限的查询语句，但是通过调用groupAuthorities ByUsername()方法，我们也能够将群组权限重写为自定义的查询语句。

将默认的SQL查询替换为自定义的设计时，很重要的一点就是要遵循查询的基本协议。所有查询都将用户名作为唯一的参数。认证查询会选取用户名、密码以及启用状态信息。权限查询会选取零行或多行包含该用户名及其权限信息的数据。群组权限查询会选取零行或多行数据，每行数据中都会包含群组ID、群组名称以及权限。

使用转码后的密码

看一下上面的认证查询，它预期用户密码存储在了数据库之中。这里唯一的问题在于如果密码明文存储，就很容易受到黑客的窃取。但是，如果数据库中的密码进行了转码，那么认证就会失败，因为它与用户提交的明文密码并不匹配。

为了解决这个问题，我们需要借助passwordEncoder()方法指定一个密码转码器（encoder）：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
          "select username, password, enabled from Users " +
          "where username=?")
      .authoritiesByUsernameQuery(
          "select username, authority from UserAuthorities " +
          "where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53cr3t");

}
```

passwordEncoder()方法可以接受Spring Security中PasswordEncoder接口的任意实现。Spring Security的加密模块包括了多个这样的实现。

- BCryptPasswordEncoder：使用bcrypt强哈希加密。
- NoOpPasswordEncoder：不进行任何转码。
- Pbkdf2PasswordEncoder：使用PBKDF2加密。
- SCryptPasswordEncoder：使用scrypt哈希加密。
- StandardPasswordEncoder：使用SHA-256哈希加密。

上述的代码中使用了StandardPasswordEncoder，但是你可以使用任意一个实现，如果内置的实现无法满足需求时，你甚至可以提供自定义的实现。PasswordEncoder接口非常简单：

```
public interface PasswordEncoder {
  String encode(CharSequence rawPassword);
  boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

不管你使用哪一个密码转码器，都需要理解一点，即数据库中的密码是永远不会解码的。用户在登录时所采取的策略与之相反，输入的密码会按照相同的算法进行转码，然后与数据库中已经转码过的密码进行对比。这个对比是在PasswordEncoder的matches()方法中进行的。

最终，我们实现了在数据库中维护Taco Cloud用户数据。但是，我并没有采用jdbcAuthentication()，因为我想到了另外一种认证方案。在介绍该方案之前，我们先看一下如何配置Spring Security依赖另一种通用的用户数据源：使用LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）访问的用户存储。

4.2.3　以LDAP作为后端的用户存储

为了配置Spring Security使用基于LDAP 认证，我们可以使用ldapAuthentication()方法。这个方法在功能上类似于jdbcAuthentication()，只不过是LDAP版本。如下的configure()方法展现了LDAP认证的简单配置：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchFilter("(uid={0})")
      .groupSearchFilter("member={0}");
}
```

方法userSearchFilter()和groupSearchFilter()用来为基础LDAP查询提供过滤条件，它们分别用于搜索用户和组。默认情况下，对于用户和组的基础查询都是空的，也就是表明搜索会在LDAP层级结构的根开始。但是我们可以通过指定查询基础来改变这个默认行为：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}");
}
```

userSearchBase()方法为查找用户提供了基础查询。同样的，groupSearchBase()为查找组指定了基础查询。我们声明用户应该在名为people的组织单元下搜索而不是从根开始，而组应该在名为groups的组织单元下搜索。

配置密码比对

基于LDAP认证的默认策略是进行绑定操作，直接通过LDAP服务器认证用户。另一种可选的方式是进行比对操作。这涉及将输入的密码

发送到LDAP目录上，并要求服务器将这个密码和用户的密码进行比对。因为比对是在LDAP服务器内完成的，实际的密码能保持私密。

如果你希望通过密码比对进行认证，可以通过声明passwordCompare()方法来实现：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare();
}
```

默认情况下，在登录表单中提供的密码将会与用户的LDAP条目中的userPassword属性进行比对。如果密码被保存在不同的属性中，可以通过passwordAttribute()方法来声明密码属性的名称：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode");
}
```

在本例中，我们指定了要与给定密码进行比对的是“passcode”属

性。另外，我们还可以指定密码转码器。在进行服务器端密码比对时，有一点非常好，那就是实际的密码在服务器端是私密的。但是进行尝试的密码还是需要通过线路传输到LDAP服务器上，这可能会被黑客所拦截。为了避免这一点，我们可以通过调用passwordEncoder()方法指定加密策略。

在前面的例子中，密码使用bcrypt密码哈希函数加密。这需要LDAP服务器上的密码也使用bcrypt进行了加密。

引用远程的LDAP服务器

到目前为止，我们忽略的一件事就是LDAP和实际的数据在哪里。我们很开心地配置Spring使用LDAP服务器进行认证，但是服务器在哪里呢？

默认情况下，Spring Security的LDAP认证假设LDAP服务器监听本机的33389端口。但是，如果你的LDAP服务器在另一台机器上，那么可以使用contextSource()方法来配置这个地址：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
```

```
        .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");
}
```

contextSource()方法会返回一个ContextSourceBuilder对象，这个对象除了其他功能以外，还提供了url()方法来指定LDAP服务器的地址。

配置嵌入式的LDAP服务器

如果你没有现成的LDAP服务器供认证使用，Spring Security还为我们提供了嵌入式的LDAP服务器。我们不再需要设置远程LDAP服务器的URL，只需通过root()方法指定嵌入式服务器的根前缀就可以了：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
        .root("dc=tacocloud,dc=com");
}
```

当LDAP服务器启动时，它会尝试在类路径下寻找LDIF文件来加载数据。LDIF（LDAP Data Interchange Format，LDAP数据交换格式）是以文本文件展现LDAP数据的标准方式。每条记录可以有一行或多行，每项包含一个name:value配对信息。记录之间通过空行进行分割。

如果你不想让Spring从整个根路径下搜索LDIF文件，那么可以通过

调用ldif()方法来明确指定加载哪个LDIF文件：

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .ldapAuthentication()
      .userSearchBase("ou=people")
      .userSearchFilter("(uid={0})")
      .groupSearchBase("ou=groups")
      .groupSearchFilter("member={0}")
      .passwordCompare()
      .passwordEncoder(new BCryptPasswordEncoder())
      .passwordAttribute("passcode")
      .contextSource()
        .root("dc=tacocloud,dc=com")
        .ldif("classpath:users.ldif");
}
```

在这里，我们明确要求LDAP服务器从类路径根目录下的users.ldif文件中加载内容。如果你比较好奇，如下就是一个包含用户数据的LDIF文件，我们可以使用它来加载嵌入式LDAP服务器：

```
dn: ou=groups,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
dn: uid=buzz,ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Buzz Lightyear
sn: Lightyear
uid: buzz
userPassword: password
dn: cn=tacocloud,ou=groups,dc=tacocloud,dc=com
objectclass: top
```

```
objectclass: groupOfNames
cn: tacocloud
member: uid=buzz,ou=people,dc=tacocloud,dc=com
```

Spring Security内置的用户存储非常便利，并且涵盖了最为常用的用户场景。但是，我们的Taco Cloud应用需要一些特殊的功能。当开箱即用的用户存储无法满足需求的时候，我们就需要创建和配置自定义的用户详情服务。

4.2.4　自定义用户认证

在上一章中，我们采用Spring Data JPA作为所有taco、配料和订单数据的持久化方案。所以，采用相同的方式来持久化用户数据是非常有意义的。如果这样做，数据最终应该位于关系型数据库之中。因此，我们可以使用基于JDBC的认证，但更好的办法是使用Spring Data repository来存储用户。

要事优先，在此之前，我们首先要创建领域对象，以及展现和持久化用户信息的repository接口。

定义用户领域对象和持久化

当Taco Cloud的顾客注册应用的时候，它们需要提供除用户名和密码之外的更多信息。他们会提供全名、地址和电话号码。这些信息可以用于各种目的，包括预先填充表单（更不用说潜在的市场销售机会）。

为了捕获这些信息，我们要创建程序清单4.5所示的User类：

程序清单4.5　定义用户实体

```
package tacos;
import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
                                          SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@RequiredArgsConstructor
public class User implements UserDetails {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  private final String username;
  private final String password;
  private final String fullname;
  private final String street;
  private final String city;
  private final String state;
  private final String zip;
  private final String phoneNumber;

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
  }

  @Override
  public boolean isAccountNonExpired() {
    return true;
  }
```

```
  @Override
  public boolean isAccountNonLocked() {
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }

  @Override
  public boolean isEnabled() {
    return true;
  }

}
```

你可能也发现了，User类要比第3章所定义的实体都更加复杂。除了定义了一些属性之外，User类还实现了Spring Security的UserDetails接口。

通过实现UserDetails接口，我们能够提供更多信息给框架，比如用户都被授予了哪些权限以及用户的账号是否可用。

getAuthorities()方法应该返回用户被授予权限的一个集合。各种is…Expired()方法要返回一个boolean值，表明用户的账号是否可用或过期。

对于User实体来说，getAuthorities()方法只是简单地返回一个集合，这个集合表明所有的用户都被授予了ROLE_USER权限。至少就现在来说，Taco Cloud没有必要禁用用户，所以所有的is…Expired()方法均返回true，表明用户是处于活跃状态的。

User实体定义完之后，我们就可以定义repository接口了：

```
package tacos.data;
import org.springframework.data.repository.CrudRepository;
import tacos.User;

public interface UserRepository extends CrudRepository<User, Long> {

  User findByUsername(String username);

}
```

除了扩展CrudRepository所得到的CRUD操作之外，UserRepository接口还定义了一个findByUsername()方法（将会在用户详情服务中用到，以便于根据用户名查找User）。

就像我们在第3章中所学到的那样，Spring Data JPA会在运行时自动生成这个接口的实现。所以，我们现在就可以编写使用该repository的用户详情接口了。

创建用户详情服务

Spring Security的UserDetailsService是一个相当简单直接的接口：

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username)
                     throws UsernameNotFoundException;
}
```

正如我们所看到的，这个接口的实现会得到一个用户的用户名，并且要么返回查找到的 UserDetails 对象，要么在根据用户名无法得到任何结果的情况下抛出 Username NotFoundException。

因为我们的User类实现了UserDetails，并且UserRepository提供了

findByUsername()方法，所以它们非常适合用在UserDetailsService实现中。程序清单4.6展现了Taco Cloud应用中将会用到的用户详情服务。

程序清单4.6　声明自定义的用户详情服务

```
package tacos.security;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.
                                              UserDetailsService;
import org.springframework.security.core.userdetails.
                                       UsernameNotFoundException;
import org.springframework.stereotype.Service;

import tacos.User;
import tacos.data.UserRepository;
@Service
public class UserRepositoryUserDetailsService
        implements UserDetailsService {

  private UserRepository userRepo;

  @Autowired
  public UserRepositoryUserDetailsService(UserRepository userRepo) {
    this.userRepo = userRepo;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
    User user = userRepo.findByUsername(username);
    if (user != null) {
      return user;
    }
    throw new UsernameNotFoundException(
                    "User '" + username + "' not found");
  }

}
```

UserRepositoryUserDetailsService通过构造器将UserRepository注入进来。然后，在loadByUsername()方法中，它调用了UserRepository的

findByUsername()方法来查找User。

loadByUsername()方法有一个简单的规则：它决不能返回null。因此，如果调用findByUsername()返回null，那么loadByUsername()将会抛出UsernameNotFoundException；否则，将会返回查找到的User。

我们注意到UserRepositoryUserDetailsService上添加了@Service。这是Spring的另外一个构造型（stereotype）注解，它表明这个类要包含到Spring的组件扫描中，所以我们不需要再明确将这个类声明为bean了。Spring将会自动发现它并将其初始化为一个bean。

但是，我们依然需要将这个自定义的用户详情服务与Spring Security配置在一起。因此，我们再次回到configure()方法：

```
@Autowired
private UserDetailsService userDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .userDetailsService(userDetailsService);

}
```

在这里，我们只是简单地调用userDetailsService()方法，并将自动装配到SecurityConfig中的UserDetailsService实例传递了进去。

像基于JDBC的认证一样，我们可以（也应该）配置一个密码转码器，这样在数据库中的密码将是转码过的。我们首先需要声明一个

PasswordEncoder类型的bean，然后通过调用passwordEncoder()方法将它注入到用户详情服务中：

```
@Bean
public PasswordEncoder encoder() {
  return new StandardPasswordEncoder("53cr3t");
}

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

  auth
    .userDetailsService(userDetailsService)
    .passwordEncoder(encoder());

}
```

我们讨论一下configure()方法中比较重要的最后一行。看上去，我们调用了encoder()方法，并将返回值传递给passwordEncoder()。实际上，encoder()方法带有@Bean注解，它将用来在Spring应用上下文中声明PasswordEncoder bean。对于encoder()的任何调用都会被拦截，并且会返回应用上下文中的bean实例。

现在，我们已经有了自定义的用户详情服务，它会通过JPA repository读取用户信息，接下来我们需要一种将用户存放到数据库中的办法。为了做到这一点，我们需要为Taco Cloud创建一个注册页面，供用户注册本应用。

注册用户

尽管在安全性方面，Spring Security会为我们处理很多事情，但是

它没有直接涉及用户注册的流程，所以我们需要借助Spring MVC的一些技能来完成这个任务。程序清单4.7所示的RegistrationController类会负责展现和处理注册表单。

程序清单4.7　用户注册控制器

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import tacos.data.UserRepository;

@Controller
@RequestMapping("/register")
public class RegistrationController {
  private UserRepository userRepo;
  private PasswordEncoder passwordEncoder;

  public RegistrationController(
      UserRepository userRepo, PasswordEncoder passwordEncoder) {
    this.userRepo = userRepo;
    this.passwordEncoder = passwordEncoder;
  }

  @GetMapping
  public String registerForm() {
    return "registration";
  }

  @PostMapping
  public String processRegistration(RegistrationForm form) {
    userRepo.save(form.toUser(passwordEncoder));
    return "redirect:/login";
  }

}
```

与很多典型的Spring MVC控制器类似，RegistrationController使用@Controller注解表明它是一个控制器，并且允许组件扫描功能发现它。

它还使用了@RequestMapping注解，这样就能处理路径为“/register”的请求了。具体来讲，对“/register”的GET请求会由registerForm()方法来处理，它只是简单地返回一个逻辑视图名registration。程序清单4.8展现了定义registration视图的Thymeleaf模板。

程序清单4.8　注册表单的Thymeleaf视图

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Register</h1>
    <img th:src="@{/images/TacoCloud.png}"/>

    <form method="POST" th:action="@{/register}" id="registerForm">

        <label for="username">Username: </label>
        <input type="text" name="username"/><br/>

        <label for="password">Password: </label>
        <input type="password" name="password"/><br/>

        <label for="confirm">Confirm password: </label>
        <input type="password" name="confirm"/><br/>

        <label for="fullname">Full name: </label>
        <input type="text" name="fullname"/><br/>

        <label for="street">Street: </label>
        <input type="text" name="street"/><br/>

        <label for="city">City: </label>
        <input type="text" name="city"/><br/>

        <label for="state">State: </label>
        <input type="text" name="state"/><br/>

        <label for="zip">Zip: </label>
```

```
        <input type="text" name="zip"/><br/>

        <label for="phone">Phone: </label>
        <input type="text" name="phone"/><br/>

        <input type="submit" value="Register"/>
    </form>

  </body>

</html>
```

当表单提交的时候，processRegistration()方法会处理HTTP POST请求。ProcessRegistration()方法得到的RegistrationForm对象绑定了请求的数据，该类的定义如下：

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import lombok.Data;
import tacos.User;

@Data
public class RegistrationForm {

  private String username;
  private String password;
  private String fullname;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String phone;

  public User toUser(PasswordEncoder passwordEncoder) {
    return new User(
        username, passwordEncoder.encode(password),
        fullname, street, city, state, zip, phone);
  }

}
```

就其大部分内容而言，RegistrationForm就是一个简单的支持

Lombok类，具有一些相关的属性。但是，toUser()方法使用这些属性创建了一个新的User对象，processRegistration()使用注入的UserRepository保存了该对象。

你肯定已经发现RegistrationController注入了一个PasswordEncoder，这其实就是我们在前面所声明的PasswordEncoder。在处理表单提交的时候，RegistrationController将其传递给toUser()方法，在将密码保存到数据库之前，会使用它对密码进行转码。通过这种方式，用户的密码可以以转码后的形式写入到数据库中，用户详情服务就能基于转码后的密码对用户进行认证了。

现在，Taco Cloud应用已经有了完整的用户注册和认证功能。但是，如果你现在启动应用，就会发现我们无法进入注册页面，也不会提示进行登录。这是因为在默认情况下，所有的请求都需要认证。接下来，我们看一下Web请求是如何被拦截和保护的，这样我们才能解决这个先有鸡还是先有蛋的诡异问题。

4.3 保护Web请求

Taco Cloud的安全性需求是用户在设计taco和提交订单之前必须要经过认证。但是，主页、登录页和注册页应该对未认证的用户开放。

为了配置这些安全性规则，需要介绍一下WebSecurityConfigurerAdapter的其他configure()方法：

```
@Override
```

```
protected void configure(HttpSecurity http) throws Exception {
  ...
}
```

configure()方法接受一个HttpSecurity对象，能够用来配置在Web级别该如何处理安全性。我们可以使用HttpSecurity配置的功能包括：

- 在为某个请求提供服务之前，需要预先满足特定的条件；
- 配置自定义的登录页；
- 支持用户退出应用；
- 预防跨站请求伪造。

配置HttpSecurity常见的需求就是拦截请求以确保用户具备适当的权限。接下来，我们会确保Taco Cloud的顾客能够满足这些需求。

4.3.1 保护请求

我们需要确保只有认证过的用户才能发起对“/design”和“/orders”的请求，而其他请求对所有用户均可用。如下的configure()实现就能实现这一点：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .hasRole("ROLE_USER")
      .antMatchers(“/”, "/**").permitAll()
    ;
}
```

对authorizeRequests()的调用会返回一个对象

（ExpressionInterceptUrlRegistry），基于它我们可以指定URL路径和这些路径的安全需求。在本例中，我们指定了两条安全规则：

- 具备ROLE_USER 权限的用户才能访问“/design”和“/orders”；
- 其他的请求允许所有用户访问。

这些规则的顺序是很重要的。声明在前面的安全规则比后面声明的规则有更高的优先级。如果我们交换这两个安全规则的顺序，那么所有的请求都会有permitAll()的规则，对“/design”和“/orders”声明的规则就不会生效了。

在声明请求路径的安全需求时，hasRole()和permitAll()只是众多方法中的两个。表4.1列出了所有可用的方法。

表**4.1**　用来定义如何保护路径的配置方法

方法	能够做什么
access(String)	如果给定的SpEL表达式计算结果为true，就允许访问
anonymous()	允许匿名用户访问
authenticated()	允许认证过的用户访问
denyAll()	无条件拒绝所有访问
	如果用户是完整认证的（不是通过Remember-me功能认证

fullyAuthenticated()	的），就允许访问
hasAnyAuthority(String...)	如果用户具备给定权限中的某一个，就允许访问
hasAnyRole(String...)	如果用户具备给定角色中的某一个，就允许访问
hasAuthority(String)	如果用户具备给定权限，就允许访问
hasIpAddress(String)	如果请求来自给定IP地址，就允许访问
hasRole(String)	如果用户具备给定角色，就允许访问
not()	对其他访问方法的结果求反
permitAll()	无条件允许访问
rememberMe()	如果用户是通过Remember-me功能认证的，就允许访问

表4.1中的大多数方法为请求处理提供了基本的安全的规则，但它们是自我限制的，也就是只能支持由这些方法所定义的安全规则。除此之外，我们还可以使用access()方法，通过为其提供SpEL表达式来声明更丰富的安全规则。Spring Security扩展了SpEL，包含了多个安全相关的值和函数，如表4.2所示。

表**4.2**　**Spring Security**对**Spring**表达式语言的扩展

安全表达式	计算结果
authentication	用户的认证对象
denyAll	结果始终为false
hasAnyRole(list of roles)	如果用户被授予了列表中任意的指定角色，结果为true
hasRole(role)	如果用户被授予了指定的角色，结果为true
hasIpAddress(IP Address)	如果请求来自指定IP，结果为true
isAnonymous()	如果当前用户为匿名用户，结果为true
isAuthenticated()	如果当前用户进行了认证，结果为true
isFullyAuthenticated()	如果当前用户进行了完整认证（不是通过Remember-me功能进行的认证），结果为true
isRememberMe()	如果当前用户是通过Remember-me自动认证的，结果为true
permitAll	结果始终为true
principal	用户的principal对象

我们可以看到，表4.2中大多数的安全规则都对应表4.1中类似的方法。实际上，借助access()方法和hasRole()、permitAll表达式，我们可以将configure()重写为程序清单4.9所示的形式：

程序清单4.9　使用Spring表达式来定义认证规则

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER')")
      .antMatchers(“/”, "/**").access("permitAll")
    ;
}
```

看上去，这似乎也没什么大不了的。毕竟，这些表达式只是模拟了我们之前通过方法调用已经完成的事情。但是，表达式可以更加灵活。例如，假设（基于某些疯狂的原因）我们只允许具备ROLE_USER权限的用户在星期二创建新taco（我们可以将其称为Taco Tuesday），这样就可以重写表达式。如下的代码展现了已修改的configure()：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER') && " +
          "T(java.util.Calendar).getInstance().get("+
          "T(java.util.Calendar).DAY_OF_WEEK) == " +
          "T(java.util.Calendar).TUESDAY")
      .antMatchers(“/”, "/**").access("permitAll")
    ;
}
```

我们可以使用SpEL实现各种各样的安全性限制。我敢打赌，你已经在想象基于SpEL所能实现的那些有趣的安全性限制了。

Taco Cloud应用的权限可以通过简单使用access()和SpEL表达式来实现，如程序清单4.9所示。现在，我们看一下如何自定义登录页以适应Taco Cloud应用的外观。

4.3.2 创建自定义的登录页

默认的登录页已经比最初丑陋的HTTP basic认证对话框好了很多，但是它依然非常简单，并且与Taco Cloud应用其他部分的外观不搭配。

为了替换内置的登录页，我们首先需要告诉Spring Security自定义登录页的路径是什么。这可以通过调用传入到configure()中的HttpSecurity对象的formLogin()方法来实现：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER')")
      .antMatchers(“/”, "/**").access("permitAll")

    .and()
      .formLogin()
        .loginPage("/login")
    ;
}
```

请注意，在调用formLogin()之前，我们通过and()方法将这一部分的配置与前面的配置连接在一起。and()方法表示我们已经完成了授权相关

的配置，并且要添加一些其他的HTTP配置。在开始新的配置区域时，我们可以多次调用and()。

在这个连接之后，我们调用formLogin()开始配置自定义的登录表单。在此之后，对loginPage()的调用声明了我们提供的自定义登录页面的路径。当Spring Security断定用户没有认证并且需要登录的时候，它就会将用户重定向到该路径。

现在，我们需要有一个控制器来处理对该路径的请求。因为我们的登录页非常简单，只有一个视图，没有其他内容，所以我们可以很简单地在WebConfig中将其声明为一个视图控制器。在映射到“/”的主页控制器基础之上，如下的addViewControllers()方法声明了登录页面的视图控制器：

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
  registry.addViewController("/").setViewName("home");
  registry.addViewController("/login");
}
```

最后，我们需要自己定义登录页的视图。我们目前使用了Thymeleaf作为模板引擎，所以如下的Thymeleaf就能实现我们的要求：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Login</h1>
    <img th:src="@{/images/TacoCloud.png}"/>
```

```
    <div th:if="${error}">
      Unable to login. Check your username and password.
    </div>

    <p>New here? Click
      <a th:href="@{/register}">here</a> to register.</p>

    <!-- tag::thAction[] -->
    <form method="POST" th:action="@{/login}" id="loginForm">
    <!-- end::thAction[] -->
      <label for="username">Username: </label>
      <input type="text" name="username" id="username" /><br/>
      <label for="password">Password: </label>
      <input type="password" name="password" id="password" /><br/>

      <input type="submit" value="Login"/>
    </form>
  </body>
</html>
```

这个登录页需要关注的事情就是表单提交到了什么地方以及用户名和密码输入域的名称。默认情况下，Spring Security会在“/login”路径监听登录请求并且预期的用户名和密码输入域的名称为username和password。但这都是可配置的，举例来说，如下的配置自定义了路径和输入域的名称：

```
.and()
  .formLogin()
    .loginPage("/login")
    .loginProcessingUrl("/authenticate")
    .usernameParameter("user")
    .passwordParameter("pwd")
```

在这里，我们声明Spring Security要监听对“/authenticate”的请求来处理登录信息的提交。同时，用户名和密码的字段名应该是user和pwd。

默认情况下，登录成功之后，用户将会被导航到Spring Security决定让用户登录之前的页面。如果用户直接访问登录页，那么登录成功之后用户将会被导航至根路径（例如，主页）。但是，我们可以通过指定默认的成功页来更改这种行为：

```
.and()
  .formLogin()
    .loginPage("/login")
    .defaultSuccessUrl("/design")
```

按照这个配置，用户直接导航至登录页并且成功登录之后将会被定向到“/design”页面。

另外，我们还可以强制要求用户在登录成功之后统一访问设计页面，即便用户在登录之前正在访问其他页面，在登录之后也会被定向到设计页面，这可以通过为defaultSuccessUrl方法传递第二个参数true来实现：

```
.and()
  .formLogin()
    .loginPage("/login")
    .defaultSuccessUrl("/design", true)
```

现在，我们已经完成了自定义的登录页面，接下来我们关注认证功能的另一面，那就是如何让用户退出应用。

4.3.3 退出

退出和应用的登录是同等重要的。为了启用退出功能，我们只需在

HttpSecurity对象上调用logout方法：

```
.and()
  .logout()
    .logoutSuccessUrl("/")
```

这样会搭建一个安全过滤器，该过滤器会拦截对“/logout”的请求。所以，为了提供退出功能，我们需要为应用的视图添加一个退出表单和按钮：

```
<form method="POST" th:action="@{/logout}">
  <input type="submit" value="Logout"/>
</form>
```

当用户点击按钮的时候，他们的session将会被清理，这样他们就退出应用了。默认情况下，用户会被重定向到登录页面，这样他们可以重新登录。但是，如果你想要将他们导航至不同的页面，那么可以调用logoutSuccessUrl()指定退出后的不同页面：

```
.and()
  .logout()
    .logoutSuccessUrl("/")
```

在本例中，用户在退出之后将会回到主页。

4.3.4 防止跨站请求伪造

跨站请求伪造（Cross-Site Request Forgery，CSRF）是一种常见的安全攻击。它会让用户在一个恶意的Web页面上填写信息，然后自动（通常是秘密的）将表单以攻击受害者的身份提交到另外一个应用上。

例如，用户看到一个来自攻击者的Web站点的表单，这个站点会自动将数据POST到用户银行Web站点的URL上（这个站点可能设计得很糟糕，无法防御这种类型的攻击），实现转账的操作。用户可能根本不知道发生了攻击，直到他们发现账号上的钱已经不翼而飞。

为了防止这种类型的攻击，应用可以在展现表单的时候生成一个CSRF token，并放到隐藏域中，然后将其临时存储起来，以便后续在服务器上使用。在提交表单的时候，token将和其他的表单数据一起发送至服务器端。请求会被服务器拦截，并与最初生成的token进行对比。如果token匹配，那么请求将会允许处理；否则，表单肯定是由恶意网站渲染的，因为它不知道服务器所生成的token。

比较幸运的是，Spring Security提供了内置的CSRF保护。更幸运的是，默认它就是启用的，我们不需要显式配置它。我们唯一需要做的就是确保应用中的每个表单都要有一个名为“_csrf”的字段，它会持有CSRF token。

Spring Security甚至进一步简化了将token放到请求的“_csrf”属性中这一任务。在Thymeleaf模板中，我们可以按照如下的方式在隐藏域中渲染CSRF token：

```
<input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

如果你使用Spring MVC的JSP标签库或者Spring Security的Thymeleaf方言，那么甚至都不用明确包含这个隐藏域（这个隐藏域会自动生成）。

在Thymeleaf中，我们只需要确保<form>的某个属性带有Thymeleaf属性前缀即可。通常这并不是什么问题，因为我们一般会使用Thymeleaf渲染相对于上下文的路径。例如，为了让Thymeleaf渲染隐藏域，我们只需要使用th:action属性就可以了：

```
<form method="POST" th:action="@{/login}" id="loginForm">
```

我们还可以禁用Spring Security对CSRF的支持，但是我很犹豫是否要为你们展现这个功能。CSRF的防护非常重要，并且很容易在表单中实现，所以我们没有理由禁用它。但是，如果你坚持要禁用，那么可以通过调用disable()来实现：

```
.and()
  .csrf()
    .disable()
```

再次强调，不要禁用CSRF防护，对于生产环境的应用来说更是如此。

Taco Cloud应用所有Web层的安全性都已经配置好了。除此之外，我们还有了一个自定义的登录页并且能够通过基于JPA的用户repository来认证用户。接下来，我们看一下如何获取已登录用户的信息。

4.4　了解用户是谁

通常，仅仅知道用户已登录是不够的，我们一般还需要知道他们是谁，这样才能优化体验。

例如，在OrderController中，在最初创建Order的时候会绑定一个订单的表单，如果我们能够预先将用户的姓名和地址填充到Order中就好了，这样用户就不需要为每个订单都重新输入这些信息了。也许更重要的是，在保存订单的时候应该将Order实体与创建该订单的用户关联起来。

为了在Order实体和User实体之间实现所需的关联，我们需要为Order类添加一个新的属性：

```
@Data
@Entity
@Table(name="Taco_Order")
public class Order implements Serializable {

...

  @ManyToOne
  private User user;

...

}
```

这个属性上的@ManyToOne注解表明一个订单只能属于一个用户，但是，一个用户却可以有多个订单。因为我们使用了Lombok，所以不需要为该属性显式定义访问器方法。

在OrderController中，processOrder()方法负责保存订单。这个方法需要修改以便于确定当前的认证用户是谁，并且要调用Order对象的setUser()方法来建立订单和用户之间的关联。

我们有多种方式确定用户是谁，常用的方式如下：

- 注入Principal对象到控制器方法中；
- 注入Authentication对象到控制器方法中；
- 使用SecurityContextHolder来获取安全上下文；
- 使用@AuthenticationPrincipal注解来标注方法。

举例来说，我们可以修改processOrder()方法，让它接受一个java.security.Principal类型的参数。然后，我们就可以使用Principal的名称从UserRepository中查找用户了：

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    Principal principal) {

...

  User user = userRepository.findByUsername(
          principal.getName());

  order.setUser(user);

...

}
```

这种方式能够正常运行，但是它会在与安全无关的功能中掺杂安全性的代码。我们可以修改processOrder()方法，让它不再接受Principal参数，而是接受Authentication对象作为参数：

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    Authentication authentication) {

...

  User user = (User) authentication.getPrincipal();
```

```
  order.setUser(user);

...

}
```

有了Authentication对象之后，我们就可以调用getPrincipal()来获取principal对象，在本例中，也就是一个User对象。需要注意，getPrincipal()返回的是java.util.Object，所以我们需要将其转换成User。

最整洁的方案可能是在processOrder()中直接接受一个User对象，不过我们需要为其添加@AuthenticationPrincipal注解，这样它才会变成认证的principal：

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    @AuthenticationPrincipal User user) {

  if (errors.hasErrors()) {
    return "orderForm";
  }

  order.setUser(user);

  orderRepo.save(order);
  sessionStatus.setComplete();

  return "redirect:/";
}
```

@AuthenticationPrincipal非常好的一点在于它不需要类型转换（前文中的Authentication则需要进行类型转换），同时能够将安全相关的代码仅仅局限于注解本身。在processOrder()得到User对象之后，我们就可以使用它并赋值给Order了。

还有另外一种方式能够识别当前认证用户是谁，但是这种方式有点麻烦，它会包含大量安全性相关的代码。我们可以从安全上下文中获取一个Authentication对象，然后像下面这样获取它的principal：

```
Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
User user = (User) authentication.getPrincipal();
```

尽管这个代码片段充满了安全性相关的代码，但是它与前文所述的其他方法相比有一个优势：它可以在应用程序的任何地方使用，而不仅仅是在控制器的处理器方法中。这使得它非常适合在较低级别的代码中使用。

4.5 小结

- Spring Security的自动配置是实现基本安全性功能的好办法，但是大多数的应用都需要显式的安全配置，这样才能满足特定的安全需求。
- 用户详情可以通过用户存储进行管理，它的后端可以是关系型数据库、LDAP或完全自定义实现。
- Spring Security会自动防范CSRF攻击。
- 已认证用户的信息可以通过SecurityContext对象（该对象可由SecurityContextHolder. getContext()返回）来获取，也可以借助@AuthenticationPrincipal注解将其注入到控制器中。

第5章　使用配置属性

本章内容：

- 细粒度的自动配置bean
- 将配置属性用到应用组件上
- 使用Spring profile

你还记得iPhone刚刚推出时的场景吗？它只是一小块由金属和玻璃组成的板子，完全不符合人们之前对于手机的认知。但是，它开创了现代智能手机的时代，完全改变了通信的方式。尽管触控手机比上一代的翻盖手机在很多方面都更加简单，功能也更强大，但是当iPhone第一次发布的时候，很难想象只有一个按钮的设备该如何用来打电话。

从某种程度上来讲，Spring Boot的自动配置与之类似。自动配置能够极大地简化Spring应用的开发。十多年来，我们都是使用Spring XML设置属性值，然后调用bean实例的setter方法，在使用自动配置之后，我们突然发现在没有显式配置的情况下，如何为bean设置属性变得不那么

显而易见了。

幸好，Spring Boot提供了配置属性（configuration property）的方法。其实，配置属性只是Spring应用上下文中bean的属性而已，它们可以通过多个源进行设置，包括JVM系统属性、命令行参数以及环境变量。

在本章中，我们暂缓实现Taco Cloud应用的新特性，将目光转向配置属性的功能。不过，当我们在后面的章节继续实现新特性时，你会发现所学的内容无疑都是有用的。我们首先看一下如何使用配置属性来微调Spring Boot的自动配置。

5.1　细粒度的自动配置

在深入了解配置属性之前，我们需要知道，在Spring中有两种不同（但相关）的配置。

- bean装配：声明在Spring应用上下文中创建哪些应用组件以及它们之间如何互相注入的配置。
- 属性注入：设置Spring应用上下文中bean的值的配置。

在Spring的XML方式和基于Java的配置中，这两种类型的配置通常会在同一个地方显式声明。在基于Java的配置中，带有@Bean注解的方法一般会同时初始化bean并立即为它的属性设置值。例如，请查看下面这个带有@Bean注解的方法，它会为嵌入式的H2数据库声明一个DataSource：

```
@Bean
public DataSource dataSource() {
  return new EmbeddedDataSourceBuilder()
      .setType(H2)
      .addScript("taco_schema.sql")
      .addScripts("user_data.sql", "ingredient_data.sql")
      .build();
}
```

在这里，addScript()和addScripts()方法设置了一些String类型的属性，它们是在数据源就绪之后要用到数据库上的SQL脚本。这就是不使用Spring Boot时我们配置DataSource bean的方法，但是借助自动配置的功能，就完全没有必要使用这种方法了。

如果在运行时类路径中能够找到H2依赖，那么Spring Boot会自动在Spring应用上下文中创建对应的DataSource bean。这个bean会运行名为schema.sql和data.sql的脚本。

但是，如果我们想要给SQL脚本使用其他的名称，该怎么办呢？或者，如果我们想要指定两个以上的SQL脚本又该怎么办呢？这就是配置属性能够发挥作用的地方了。但是，在开始使用配置属性之前，我们需要理解这些属性是从哪里来的。

5.1.1 理解Spring的环境抽象

Spring的环境抽象是各种配置属性的一站式服务。它抽取了原始的属性，这样需要这些属性的bean就可以从Spring本身中获取了。Spring环境会拉取多个属性源，包括：

- JVM系统属性；
- 操作系统环境变量；
- 命令行参数；
- 应用属性配置文件。

它会将这些属性聚合到一个源中，通过这个源可以注入到Spring的bean中。图5.1阐述了来自各个属性源的属性是如何流经Spring的环境抽象进入Spring bean的。

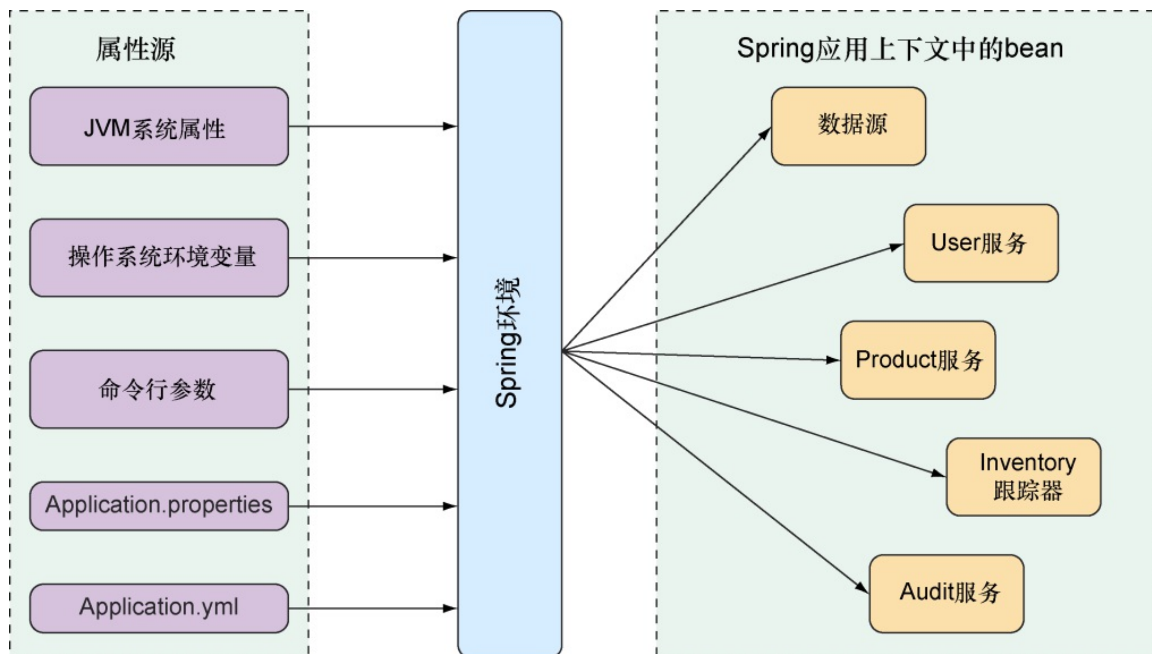


图5.1　Spring环境从各个属性源拉取属性，并让Spring应用上下文中的bean可以使用它们

Spring Boot自动配置的bean都可以通过Spring环境提取的属性进行配置。举个简单的例子，假设我们希望应用底层的Servlet容器使用另外一个端口监听请求，而不再使用8080。为了实现这一点，我们可以在“src/main/resources/application.properties”中将server.port设置成一个不同的端口，如下所示：

```
server.port=9090
```

在设置属性的时候，我个人更喜欢使用YAML。所以，我通常不会使用application.properties，而是在“src/main/resources/application.yml”中设置server.port的值，如下所示：

```
server:
  port: 9090
```

如果你喜欢在外部配置该属性，那么可以在使用命令行参数启动应用的时候指定端口：

```
$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090
```

如果你希望应用始终在一个特定的端口启动，那么可以通过操作系统的环境变量进行一次性的设置：

```
$ export SERVER_PORT=9090
```

需要注意，在将属性设置为环境变量的时候，命名风格略有不同，这样做是为了适应操作系统对环境变量名称的限制。不过，没有关系，Spring能够将其挑选出来，并将SERVER_PORT解析为server.port。

正如我前面所说，有多种配置属性的方法。当我们学习第14章的时候，你会看到另外一种设置属性的方法，那就是通过中心化的配置服务器实现。实际上，我们可以使用几百个配置属性来调整Spring bean的行为。你已经看到了其中的一部分，比如本章中已经介绍的server.port。

在本章中，我们不可能介绍所有可用的配置属性。尽管如此，我们

还是可以了解一些你可能会经常遇到的非常有用的配置属性。我们首先看一下能够调整自动配置的数据源的一些属性。

5.1.2 配置数据源

此时，Taco Cloud应用尚未完成，在该应用准备部署之前，我们还有好几个章节来完善它。因此，使用嵌入式的H2数据库作为数据源非常适合我们的需求，至少就目前来看是这样的。但是，一旦要将应用部署到生产环境中，你可能需要考虑一个更加持久的数据库解决方案。

尽管我们可以显式地配置自己的DataSource，但通常没有必要这样做。相反，通过配置属性设置数据库URL和凭证信息会更加简单。例如，如果你想要开始使用MySQL数据库，那么可以把如下的配置属性添加到application.yml中：

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
username: tacodb
password: tacopassword
```

尽管我们需要将对应的JDBC驱动添加到构建文件中，但是我们不需要指定JDBC驱动类。Spring Boot会根据数据库URL的结构推算出来。然而，如果这样做有问题的话，我们依然可以通过spring.datasource.driver-class-name属性来进行设置：

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
```

```
    username: tacodb
    password: tacopassword
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot在自动化配置DataSource bean的时候，会使用该连接。如果在类路径中存在Tomcat的JDBC连接池，DataSource将使用该连接池。否则，Spring Boot将会在类路径下尝试查找并使用如下的连接池实现：

- HikariCP
- Commons DBCP 2

自动配置所能支持的连接池可选方案仅有这些，但是随时欢迎显式配置DataSource bean，这样你可以使用任意喜欢的连接池实现。

在本章前面的内容中，我们建议要有一种方式声明应用启动的时候要执行的数据库初始化脚本。在这种情况下，spring.datasource.schema和spring.datasource.data属性就非常有用了：

```
spring:
  datasource:
    schema:
    - order-schema.sql
    - ingredient-schema.sql
    - taco-schema.sql
    - user-schema.sql
    data:
    - ingredients.sql
```

有的读者可能无法使用显式配置数据源的方式，而是更加倾向于在JNDI中配置数据源并让Spring去那里进行查找。在这种情况下，我们可以使用spring.datasource.jndi-name搭建自己的数据源：

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/tacoCloudDS
```

如果我们设置了spring.datasource.jndi-name属性，其他的数据库连接属性（已经设置了的话）就会被忽略掉。

5.1.3　配置嵌入式服务器

我们已经看到过如何使用server.port属性来配置servlet容器的端口。但是，我还没有展示将server.port设置为0将会出现什么状况：

```
server:
  port: 0
```

尽管我们将server.port属性显式设置成了0，但是服务器并不会真的在端口0上启动。相反，它会任选一个可用的端口。在我们运行自动化集成测试的时候，这会非常有用，因为这样能够保证并发运行的测试不会与硬编码的端口号冲突。在第 13 章中我们将会看到，如果不关心应用在哪个端口启动，那么这种配置方式也非常有用，因为此时应用将会变成通过服务注册中心来进行查找的微服务。

但是，底层服务器的配置并不仅仅局限于一个端口，我们对底层容器常见的一项设置就是让它处理HTTPS请求。为了实现这一点，我们首先要使用JDK的keytool命令行工具生成keystore：

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

在这个过程中，会询问我们一些关于名称和组织机构相关的问题，大多数问题都无关紧要。但是，它提示输入密码的时候需要记住你所选择的密码。在本例中，我选择使用letmein作为密码。

接下来，我们需要设置一些属性，以便于在嵌入式服务器中启用HTTPS。我们可以在命令行中进行配置，但是这种方式非常不方便，相反，你可能更愿意通过application.properties或application.yml文件来声明配置。在application.yml中，配置属性如下所示：

```
server:
  port: 8443
  ssl:
    key-store: file:///path/to/mykeys.jks
    key-store-password: letmein
    key-password: letmein
```

在这里，我们将server.port设置为8443，这是在开发阶段HTTPS服务器的常用选择。server.ssl.key-store属性应该设置为我们所创建的keystore文件的路径。在这里，它使用了file:// URL，因此会在文件系统中加载，但是，如果你需要将它打包到一个应用JAR文件中，就需要使用“classpath:”URL来引用它。server.ssl.key-store-password和server.ssl.key-password属性都设置成了创建keystore时所设置的密码。

这些属性准备就绪之后，应用就会监听8443端口上的HTTPS请求。因为浏览器之间有所差异，所以你可能会遇到服务器无法验证其身份的警告。在开发阶段，通过localhost提供服务时，这其实无须担心。

5.1.4 配置日志

大多数的应用都会提供某种形式的日志。即便你的应用本身不直接打印任何日志，应用所使用的库肯定也会以日志的形式记录它们的活动。

默认情况下，Spring Boot通过Logback配置日志，日志会以INFO级别写入到控制台中。在运行应用或其他样例的时候，你可能已经在应用日志中发现了大量的INFO级别的条目。

为了完全控制日志的配置，我们可以在类路径的根目录下（在src/main/resources中）创建一个logback.xml文件。如下是一个简单logback.xml文件的样例：

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <logger name="root" level="INFO"/>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

除了日志所使用的模式之外，这个Logback配置和没有logback.xml文件时的默认行为几乎是相同的。但是，通过编辑logback.xml文件，我们可以完全控制应用的日志文件。

注意：关于logback.xml文件中都可以声明哪些内容，这超出

了本书的范围。你可以参考Logback的文档来了解更多信息。

在日志配置方面，你可能遇到的常见变更就是修改日志级别和指定日志写入到哪个文件中。借助Spring Boot的配置属性功能，我们不用创建logback.xml文件就能完成这些变更。

要设置日志级别，我们可以创建以logging.level作为前缀的属性，随后紧跟着的是我们想要设置日志级别的logger。假设，我们想要将root logging设置为WARN级别，但是希望将Spring Security的日志级别设置为DEBUG。那么，在application.yml中添加如下的条目就能实现我们的要求：

```
logging:
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

我们还可以将Spring Security的包名扁平化到一行中，使其更易于阅读：

```
logging:
  level:
    root: WARN
    org.springframework.security: DEBUG
```

现在，假设我们想要将日志条目写入到“/var/logs/”中的TacoCloud.log文件中。logging.path和logging.file文件可以按照如下形式

进行设置：

```
logging:
  path: /var/logs/
  file: TacoCloud.log
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

假设应用具有“/var/logs/”目录的写入权限，那么日志条目会写入到“/var/logs/ TacoCloud.log”文件中，默认情况下，日志文件一旦达到10MB，就会轮换。

5.1.5　使用特定的属性值

在设置属性的时候，我们并非必须要将它们的值设置为硬编码的String或数值。其实，我们还可以从其他的配置属性派生值。

例如，假设（不管基于什么原因）我们想要设置一个名为greeting.welcome的属性，它的值来源于名为spring.application.name的另一个属性。为了实现该功能，在设置greeting.welcome的时候，我们可以使用${}占位符标记：

```
greeting:
  welcome: ${spring.application.name}
```

我们甚至可以将占位符嵌入到其他文本中：

```
greeting:
```

```
welcome: You are using ${spring.application.name}.
```

我们可以看到，在配置Spring自己的组件时，使用配置属性可以很容易地将值注入这些组件属性中，并且可以细粒度地调整自动配置功能。配置属性并不专属于Spring创建的bean。我们稍微下点功夫就可以在自己的bean中使用配置属性功能。接下来，让我们看一下如何实现。

5.2　创建自己的配置属性

正如我在前文所述，配置属性只不过是bean的属性，它们可以从Spring的环境抽象中接受配置。我还没有提及的是这些bean该如何消费这些配置。

为了支持配置属性的注入，Spring Boot提供了@ConfigurationProperties注解。将它放到Spring bean上之后，它就会为该bean中那些能够根据Spring环境注入值的属性赋值。

为了阐述@ConfigurationProperties是如何运行的，假设我们为OrderController添加了如下的方法，该方法会列出当前认证用户过去的订单：

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user));

  return "orderList";
}
```

除此之外，我们还要为OrderRepository添加必要的findByUser()方法：

```
List<Order> findByUserOrderByPlacedAtDesc(User user);
```

注意，这个repository方法使用了OrderByPlacedAtDesc子句。OrderBy区域指定了结果要按照什么属性来排序，在本例中，也就是placedAt属性。最后的Desc声明要按照降序进行排列。所以，返回的订单将会按照时间由近及远进行排序。

按照这种写法，如果用户只创建了少量订单，那么这个控制器方法可能会非常有用，但是，对于狂热的taco爱好者来说，这种方式就显得有些不方便了。在浏览器中显示一些订单会很有用，但是一长串没完没了的订单列表简直就是“噪声”。假设，我们希望将显示的订单数量限制为最近的20个，那么我们可以按照如下方式来修改ordersForUser()：

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  Pageable pageable = PageRequest.of(0, 20);
  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

  return "orderList";
}
```

OrderRepository也需要对应修改：

```
List<Order> findByUserOrderByPlacedAtDesc(
        User user, Pageable pageable);
```

现在，我们修改了findByUserOrderByPlacedAtDesc()方法的签名，使其能够接受Pageable参数。Pageable是Spring Data根据页号和每页数量选取结果的子集的一种方法。在ordersForUser()控制器方法中，我们构建了一个PageRequest对象，该对象实现了Pageable，我们将其声明为请求第一页（序号为0）的数据，并且每页数量为20，这样我们就能获取当前用户最近的20个订单。

尽管这种方式能够很好地运行，但是我们在这里硬编码了每页的数量，这有点让人担心。如果我们以后发现展示20个订单太多，并决定将其修改为10个，那该怎么办？因为这个值是硬编码的，所以需要重新构建和重新部署应用。

我们可以将每页数量设置成一个自定义的配置属性，而不是硬编码到代码中。首先，我们需要添加一个名为pageSize的新属性到OrderController中，并为OrderController添加@ConfigurationProperties注解，如程序清单5.1所示。

程序清单5.1　在OrderController中启用配置属性功能

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
@ConfigurationProperties(prefix="taco.orders")
public class OrderController {

  private int pageSize = 20;

  public void setPageSize(int pageSize) {
    this.pageSize = pageSize;
  }

  ...
```

```
  @GetMapping
  public String ordersForUser(
      @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, pageSize);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));
    return "orderList";
  }

}
```

程序清单5.1最重要的变更是添加了@ConfigurationProperties注解。它的prefix属性设置成了taco.orders，这意味着当设置pageSize的时候，我们需要使用名为taco.orders.pageSize的配置属性。

新的pageSize值默认为20，但是通过设置taco.orders.pageSize属性，我们可以很容易地将其修改为任意的值。例如，我们可以在application.yml中按照如下的方式设置该属性：

```
taco:
  orders:
    pageSize: 10
```

如果在生产环境中需要快速更改，我们可以将taco.orders.pageSize设置为环境变量，这样就不用重新构建和重新部署应用了：

```
$ export TACO_ORDERS_PAGESIZE=10
```

设置配置属性的任何方式都可以用来调整最近订单页面中每页的数量。接下来，我们看一下如何在属性持有者（property holder）中设置配置数据。

5.2.1　定义配置属性的持有者

这里并没有说@ConfigurationProperties只能用到控制器或特定类型的bean中。@ConfigurationProperties实际上通常会放到一种特定类型的bean中，这种bean的目的就是持有配置数据。这样的话，特定的配置细节就能从控制器和其他应用程序类中抽离出来，多个bean也能更容易地共享一些通用的配置。

针对OrderController中的pageSize属性，我们可以将其抽取到一个单独的类中。程序清单5.2就以这样的方式来使用OrderProps类。

程序清单5.2　将pageSize抽取到持有者类中

```
package tacos.web;
import org.springframework.boot.context.properties.
                                        ConfigurationProperties;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {

  private int pageSize = 20;

}
```

就像我们在OrderController中所做的那样，pageSize的默认值为20，OrderProps使用了@ConfigurationProperties注解并且将前缀设置成了taco.orders。这个类还用到了@Component注解，这样Spring的组件扫描功能会自动发现它并将其创建为Spring应用上下文中的bean。这是非常

重要的，因为我们下一步要将OrderProps作为bean注入到OrderController中。

配置属性持有者并没有什么特别之处。它们只是将Spring环境注入到其属性中的bean。它们可以注入到任意需要这些属性的其他bean中。对于OrderController来说，我们就可以从OrderController中移除pageSize，并注入和使用OrderProps bean：

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
public class OrderController {

  private OrderRepository orderRepo;

  private OrderProps props;

  public OrderController(OrderRepository orderRepo,
          OrderProps props) {
    this.orderRepo = orderRepo;
    this.props = props;
  }

    ...

  @GetMapping
  public String ordersForUser(
      @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, props.getPageSize());
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
  }

    ...

}
```

现在，OrderController不需要负责处理自己的配置属性了。这样能够让OrderController中的代码更加整洁一些，并且能够让其他的bean重用OrderProps中的属性。除此之外，我们可以将订单相关的属性全部放到一个地方，也就是OrderProps类中。如果我们需要添加、删除、重命名或者以其他方式更改其中的属性，我们只需要在OrderProps中进行变更就可以了。

例如，假设我们在多个其他的bean中也用到了pageSize属性，现在我们决定要对这个属性的值进行一些校验，限制它的值必须要不小于5且不大于25。如果没有持有者bean，我们必须要将校验注解用到OrderController的pageSize属性上以及其他所有使用该属性的类上。但是，因为我们现在将pageSize抽取到了OrderProps中，所以只需要修改OrderProps就可以了：

```
package tacos.web;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.
                                        ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
@Validated
public class OrderProps {

  @Min(value=5, message="must be between 5 and 25")
  @Max(value=25, message="must be between 5 and 25")
  private int pageSize = 20;
```

```
}
//end::validated[]
```

尽管我们很容易就可以将@Validated、@Min和@Max注解用到OrderController（和其他可以注入OrderProps的地方），但是这样会使OrderController更加混乱。通过配置属性的持有者bean，我们将所有的配置属性收集到了一个地方，这样就能让使用这些属性的bean尽可能保持整洁。

5.2.2　声明配置属性元数据

在IDE中，你可能会发现application.yml（或application.properties）文件的taco.orders. pageSize条目上会有一条警告信息，根据IDE不同显示会有所差异，这个警告提示的内容可能是“Unknown property ‘taco’”。这个警告产生的原因在于我们刚刚创建的配置属性缺少元数据。图5.2展示了在Spring Tool Suite中，当我将鼠标悬停到taco属性时的样式。

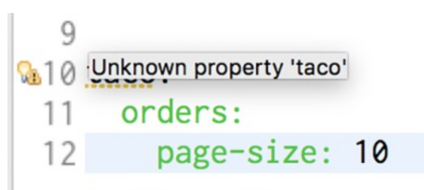


图5.2　缺少配置属性元数据所产生的警告

配置属性的元数据完全是可选的，它并不会妨碍配置属性的运行。但是，元数据对于为配置属性提供一个最小化的文档非常有用，在IDE中尤为如此。

举例来说，将鼠标指针悬停到security.user.password属性上时，就会看到图5.3那样的效果。尽管悬停对我们的帮助很有限，但是它足以让我们知道这个属性是做什么的以及如何使用它。

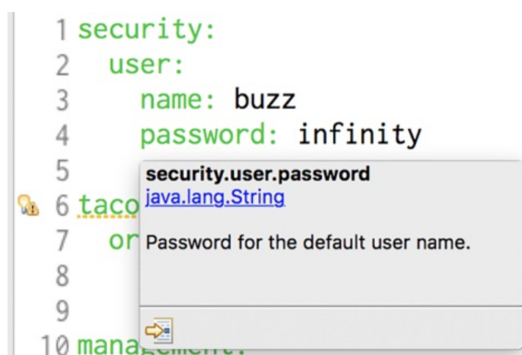


图5.3 Spring Tool Suite中配置属性的悬停文档

为了帮助那些使用你所定义的配置属性的人（有可能就是你本人），为这些属性创建一些元数据是非常好的办法，至少它能消除IDE上那些烦人的黄色警告。

为了创建自定义配置属性的元数据，我们需要在META-INF下创建一个名为additional-spring-configuration-metadata.json的文件（比如，在项目的“src/main/resources/ META-INF”目录下）。

快速添加缺失的元数据

如果使用Spring Tool Suite，就会有一个创建缺失属性元数据的快速修正选项。将鼠标放到缺失元数据警告的那行代码上，在Mac下按CMD+1组合键或者在Windows和Linux下按Ctrl+1组合键就能打开快速修正的弹出框（见图5.4）。

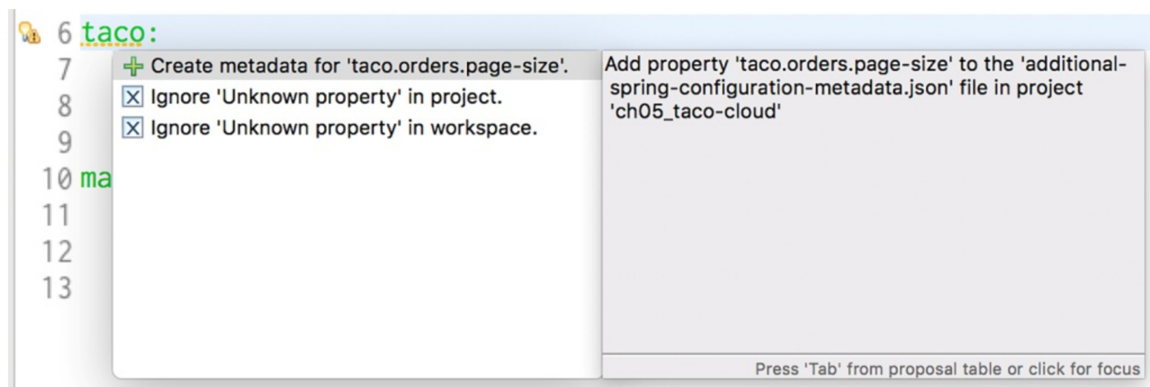


图5.4　在Spring Tool Suite中通过快速修正弹出框创建配置属性

然后，选择“Create Metadata for ...”选项来为属性添加元数据（会在additional-spring- configuration-metadata.json文件中进行添加），如果文件还不存在，将会自动创建该文件。

对于taco.orders.pageSize属性来说，我们可以通过如下的JSON为其添加元数据：

```
{
  "properties": [
    {
      "name": "taco.orders.page-size",
      "type": "java.lang.String",
      "description":
          "Sets the maximum number of orders to display in a list."
    }
  ]
}
```

需要注意，在元数据中引用的属性名为taco.orders.page-size。Spring Boot灵活的属性命名功能允许属性名出现不同的变种，比如taco.orders.page-size等价于taco.orders.pageSize。

元数据准备就绪之后，警告信息就会消失了。除此之外，如果将鼠

标指针悬停到taco.orders. pageSize属性上，就会看到如图5.5所示的描述信息。

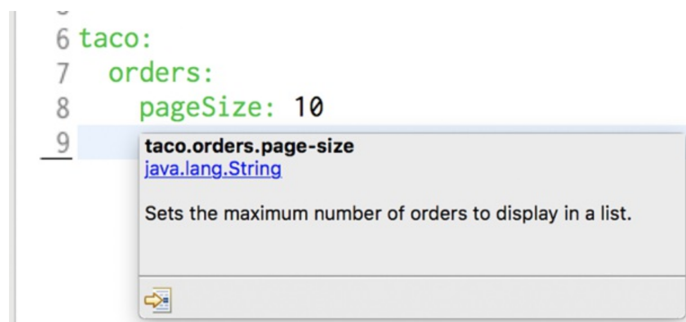


图5.5　自定义配置属性的悬停帮助信息

另外，在IDE中，就像Spring本身提供的配置属性一样，我们还能具备自动补全功能，如图5.6所示。

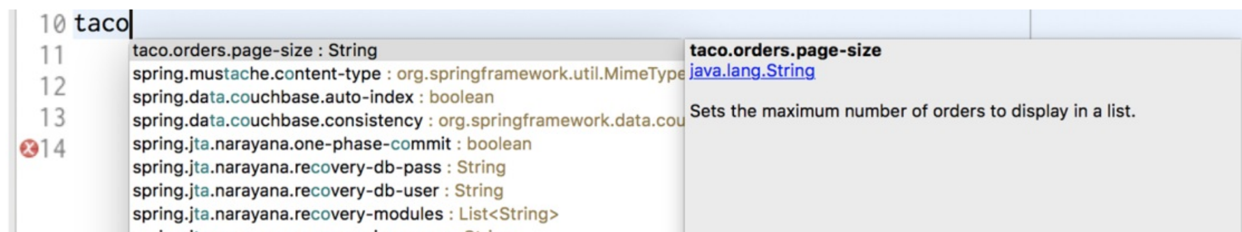


图5.6　配置属性的元数据能够帮助实现属性的自动补全功能

我们可以看到，配置属性对于调整自动配置的组件以及应用程序自身的bean都非常有用。但是，如果我们想要为不同的部署环境配置不同的属性又该怎么办呢？接下来，我们看一下该如何使用Spring profile搭建特定环境的配置。

5.3　使用profile进行配置

当应用部署到不同的运行时环境中的时候，有些配置细节通常会有些差别。例如，数据库连接的细节在开发环境和质量保证（quality assurance）环境中可能就不相同，而它们与生产环境可能又不一样。配置不同环境之间有差异的属性时，有种办法就是使用环境变量，通过这种方式来指定配置属性，而不是在application.properties和application.yml中进行定义。

例如，在开发阶段，我们可以依赖自动配置的嵌入式H2数据库。但是在生产环境中，我们可以按照如下的方式将数据库配置属性设置为环境变量：

```
% export SPRING_DATASOURCE_URL=jdbc:mysql://localhost/tacocloud
% export SPRING_DATASOURCE_USERNAME=tacouser
% export SPRING_DATASOURCE_PASSWORD=tacopassword
```

尽管这种方式可以运行，但是如果配置属性比较多，那么将它们声明为环境变量会非常麻烦。除此之外，我们没有好的方式来跟踪环境变量的变化，也无法在出现错误的时候进行回滚。

相对于这种方式，我更加倾向于采用Spring profile。profile是一种条件化的配置，在运行时，根据哪些profile处于激活状态，可以使用或忽略不同的bean、配置类和配置属性。

例如，为了开发和调试方便，我们希望使用嵌入式的H2数据库，并且Taco Cloud代码的日志级别为DEBUG。但是在生产环境中，我们希望使用外部的MySQL数据库，并将日志级别设置为WARN。在开发场景下，我们可以很容易地设置数据源属性并使用自动配置的H2数据

库。对于调试级别的日志需求，我们可以在application.yml文件中通过logging.level.tacos属性将tacos基础包的日志级别设置为DEBUG：

```
logging:
  level:
    tacos: DEBUG
```

这就是我们要针对开发环境做的事情。但是，如果我们不对application.yml做任何修改就将应用部署到生产环境，tacos包依然会写入调试日志并且依然会使用H2数据库。我们需要做的就是定义一个profile，其中包含适用于生产环境的属性。

5.3.1　定义特定profile的属性

定义特定profile相关的属性的一种方式就是创建另外一个YAML或属性文件，其中只包含用于生产环境的属性。文件的名称要遵守如下的约定：application-{profile名}.yml或 application-{profile名}.properties。然后，我们就可以在这里声明适用于该profile的配置属性了。例如，我们可以创建一个新的名为application-prod.yml的文件，其中包含如下属性：

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
logging:
  level:
    tacos: WARN
```

定义特定profile相关的属性的另外一种方式仅适用于YAML配置。它会将特定profile的属性和非profile的属性都放到application.yml中，它们之间使用3个中划线进行分割，并且使用spring.profiles属性来命名profile。如果按照这种方式定义生产环境的属性，等价的application.yml如下所示：

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

logging:
  level:
    tacos: WARN
```

我们可以看到，application.yml文件通过一组中划线（---）分成了两部分。第二部分指定了spring.profiles值，代表后面的属性适用于prod profile。而第一部分的属性没有指定spring.profiles，所以它们是所有profile通用的，或者如果当前激活的profile没有设置这些属性，它们就会作为默认值。

不管应用程序运行的时候哪个profile处于激活状态，根据默认profile，tacos包的日志级别都将会设置为DEBUG。但是，如果名为prod的profile激活，那么logging.level.tacos属性将会被重写为WARN。与之类似，如果prod profile处于激活状态，那么数据源相关的属性将会被设

置为使用外部的MySQL数据库。

通过创建模式为application-{profile名}.yml或application-{profile名}.properties的YAML或属性文件，我们可以按需定义任意数量的profile。或者，我们也可以在application.yml中再输入3个中划线，结合spring.profiles属性来指定其他名称的profile，然后添加该profile特定的相关属性。

5.3.2 激活profile

如果我们不激活这些profile，声明profile相关的属性其实没有任何用处。但是，我们该如何激活一个profile呢？要激活某个profile，需要做的就是将profile名称的列表赋值给spring.profiles.active属性。例如，在application.yml中，我们可以这样设置：

```
spring:
  profiles:
    active:
    - prod
```

但是，这可能是激活profile最糟糕的一种方式。如果我们在application.yml中设置处于激活状态的profile，那么这个profile就会变成默认的profile，我们体验不到使用profile将生产环境相关属性和开发环境相关的属性分开的任何好处。因此，我推荐使用环境变量来设置处于激活状态的profile。在生产环境中，我们可以这样设置SPRING_PROFILES_ACTIVE：

```
% export SPRING_PROFILES_ACTIVE=prod
```

这样部署到该机器上的任何应用就都会激活prod profile，对应的属性会比默认profile具备更高的优先级。

如果以可执行JAR文件的形式运行应用，那么我们还可以以命令行参数的形式设置激活的profile：

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

你可能已经注意到了，spring.profiles.active属性名是复数形式的profile。这意味着我们可以设置多个激活的profile。如果使用环境变量，通常这可以通过逗号分隔的列表来实现：

```
% export SPRING_PROFILES_ACTIVE=prod,audit,ha
```

但是，在YAML中，我们要按照如下的方式来声明列表：

```
spring:
  profiles:
    active:
    - prod
    - audit
    - ha
```

另外，值得一提的是，如果我们将Spring应用部署到Cloud Foundry中，将会自动激活一个名为cloud的profile。如果你的生产环境是Cloud Foundry，那么你可以将生产环境相关的属性放到cloud profile下。

在Spring应用中，profile不仅能够用来条件化地设置配置属性，接下来我们看一下如何基于处于激活状态的profile来声明特定的bean。

5.3.3 使用profile条件化地创建bean

有时候，为不同的profile创建一组独特的bean是非常有用的。正常情况下，不管哪个profile处于激活状态，Java配置类中声明的所有bean都会被创建。但是，假设我们希望某些bean仅在特定profile激活的情况下才需要创建。在这种情况下，@Profile注解可以将某些bean设置为仅适用于给定的profile。

例如，在TacoCloudApplication中，我们有一个CommandLineRunner bean，它用来在应用启动的时候加载嵌入式数据库和配料数据。对于开发阶段来讲，这是很不错的；但是对于生产环境的应用来说，就没有必要（也是不符合需求的）了。为了防止在生产部署环境中每次都加载配料数据，我们可以为声明CommandLineRunner bean的方法添加@Profile注解，如下所示：

```
@Bean
@Profile("dev")
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {

  ...

}
```

或者，假设我们在dev或qa profile激活的时候都需要创建CommandLineRunner。在这种情况下，我们可以为要创建的bean把所有profile都列出来：

```
@Bean
@Profile({"dev", "qa"})
```

```
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {
  ...
}
```

现在，配料数据会在dev或qa profile激活的时候才加载。这意味着，在开发环境运行的时候我们需要将dev profile激活。如果除了prod激活时，CommandLineRunner bean都需要创建，那么我们可以采用一种更简便的方式。在这种情况下，我们可以按照如下的方式来使用@Profile：

```
@Bean
@Profile("!prod")
public CommandLineRunner dataLoader(IngredientRepository repo,
      UserRepository userRepo, PasswordEncoder encoder) {

  ...

}
```

在这里，感叹号（!）否定了profile的名称。实际上，它的含义是只要prod profile不激活就要创建CommandLineRunner bean。

我们还可以在带有@Configuration注解的类上使用@Profile。例如，假设我们要将CommandLineRunner抽取到一个名为DevelopmentConfig的配置类中，那么我们可以按照如下的方式为DevelopmentConfig添加@Profile：

```
@Profile({"!prod", "!qa"})
@Configuration
public class DevelopmentConfig {

  @Bean
  public CommandLineRunner dataLoader(IngredientRepository repo,
        UserRepository userRepo, PasswordEncoder encoder) {
```

```
    ...
  }
}
```

在这里，CommandLineRunner bean（包括DevelopmentConfig中定义的其他bean）只有在prod和qa均没有激活的情况下才会创建。

5.4　小结

- Spring bean可以添加@ConfigurationProperties注解，这样就能够从多个属性源中选取一个来注入它的值。
- 配置属性可以通过命令行参数、环境变量、JVM系统属性、属性文件或YAML文件等方式进行设置。
- 配置属性可以用来覆盖自动配置相关的设置，包括指定数据源URL和日志级别。
- Spring profile可以与属性源协同使用，从而能够基于激活的profile条件化地设置配置属性。

第2部分　Spring集成

第2部分的章节将会涵盖Spring应用与其他应用集成的话题。

第6章将扩展第2章对Spring MVC的讨论，介绍如何在Spring中编写REST API。我们将会看到如何使用Spring MVC定义REST端点、启用超媒体REST资源以及使用Spring Data REST自动生成基于repository的REST端点。第7章转换视角，关注Spring应用如何消费REST API的话题。在第8章中，我们将会学习如何借助异步通信技术让Spring发送和接收Java Message Service （JMS）、RabbitMQ与Kafka的消息。在最后的第9章中，我们将探讨使用Spring Integration项目实现声明式应用集成的话题。我们会涵盖实时处理数据、定义集成流以及与外部系统（如Email和文件系统）集成的功能。

第6章　创建REST服务

本章内容：

- 在Spring MVC中定义REST端点
- 启用超链接REST资源
- 自动化基于repository的REST端点

“Web浏览器已死，那么现在是谁的天下呢？”

十多年前，我就听到有人说Web浏览器已经行将就木，它会被其他的事物所取代。但是，这怎么可能会实现呢？谁有可能取代几乎无处不在的Web浏览器呢？如果没有Web浏览器，我们该如何消费越来越多的网络站点和在线服务呢？这肯定是某个疯子的胡言乱语！

我们快进到今天，显然，Web浏览器并没有消失，但它已经不再是访问互联网的主要方式了。现在，移动设备、平板电脑、智能手表和基于语音的设备已经非常常见，甚至很多基于浏览器的应用实际上运行的都是JavaScript应用，而不再是让浏览器成为服务器渲染内容的哑终端。

随着客户端的可选方案越来越多，许多应用程序采用了一种通用的设计，那就是将用户界面推到更接近客户端的地方，而让服务器公开API，通过这种API，各种客户端都能与后端功能进行交互。

在本章中，我们将会使用Spring来为Taco Cloud应用提供REST API。在这里我们将会用到第2章中已经学习过的Spring MVC，使用Spring MVC的控制器创建RESTful端点。同时，我们还会将第4章中定义的Spring Data repository暴露为REST端点。最后，我们将会看一下如何测试和保护这些端点。首先，我们需要编写几个新的Spring MVC控制器，它们会使用REST端点来暴露后端功能，这些端点将会被富Web前端所消费。

6.1　编写RESTful控制器

当你翻看本章并阅读简介时，就会发现我重新设想了Taco Cloud的用户界面，希望你不要介意。你之前的工作成果可能比较适合起步，但是在美学方面也许会有所欠缺。

图6.1是新的Taco Cloud外观的示例，看上去很时尚吧？

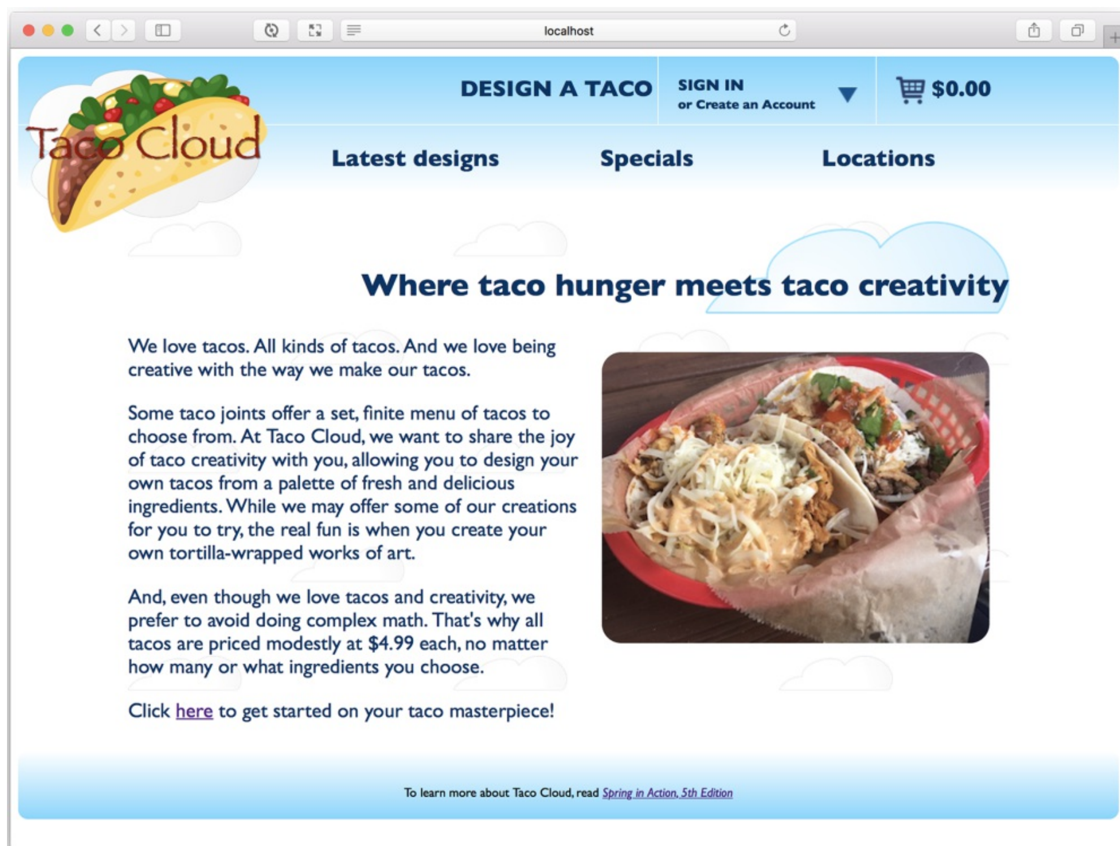


图6.1　新的Taco Cloud主页

在改善Taco Cloud外观的同时，我还决定使用流行的Angular框架将前端构建为单页应用。最终，这个新的浏览器UI将替换我们在第2章中创建的服务器渲染页面。但是，想要实现这一点，我们需要创建一个REST API，基于Angular[1]的UI将会与之通信，以保存和获取taco数据。

是否要采用**SPA**？

在第2章中，我们使用Spring MVC开发了一个传统的多页应用（MultiPage Application，MPA），现在我们要将其替换为基于Angular的单页应用（Single-Page Application，SPA）。但是，

我并不认为SPA始终是比MPA更好的可选方案。

在SPA中，展现和后端处理在很大程度上是解耦的，这样就提供了为相同的后端功能开发多个用户界面（例如原生移动应用）的机会。它还为与其他可以使用API的应用程序集成创造了可能性。但并不是所有的应用程序都需要这种灵活性，如果你只需要在Web页面上显示信息，那么MPA是一种更简单的设计。

这并不一本关于Angular的书，所以本章中的代码将会主要关注后端的Spring代码。我只会给出适当的Angular代码，以便于让你了解客户端是如何运行的。但是，请放心，完整的代码集会包括Angular前端，它们都是本书配套代码的一部分。如果你有兴趣，可以阅读Jeremy Wilken编写的*Angular in Action*（Manning，2018）以及Yakov Fain和Anton Moiseev编写的*Angular Development with TypeScript, Second Edition*（Manning，2018）。

本质上来讲，Angular客户端代码将会通过HTTP请求与本章所创建的API进行通信。在第2章中，我们使用@GetMapping注解从服务端获取数据，使用@PostMapping注解往服务器端提交数据。在定义REST API的时候，这些注解依然有用。除此之外，Spring MVC还为各种类型的HTTP请求提供了一些其他的注解，如表6.1所示。

表**6.1　Spring MVC**的**HTTP**请求处理注解

注解	**HTTP**方法	典型用途[a]

@GetMapping	HTTP GET请求	读取资源数据
@PostMapping	HTTP POST请求	创建资源
@PutMapping	HTTP PUT请求	更新资源
@PatchMapping	HTTP PATCH请求	更新资源
@DeleteMapping	HTTP DELETE请求	删除资源
@RequestMapping	通用的请求处理，HTTP方法可以通过method声明	

[a] 将HTTP方法映射为创建、读取、更新和删除（CRUD）操作并不是非常恰当，但是在实践中这是常见的使用方式，在我们的Taco Cloud应用中也是这样使用它们的。

要实际看到这些注解的效果，我们需要创建一个简单的REST端点，该端点会检索一些最新创建的taco。

6.1.1 从服务器中检索数据

Taco Cloud应用最酷的一件事就是它允许taco迷设计自己的taco作品，并与taco爱好者分享他们的作品。为此，Taco Cloud需要能够在单击“Latest Designs”链接时显示最近创建的taco列表。

在Angular代码中，我定义了RecentTacosComponent组件，它会展现

最新创建的taco。RecentTacosComponent完整的TypeScript代码如程序清单6.1所示。

程序清单6.1　展现最近taco的Angular组件

```
import { Component, OnInit, Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'recent-tacos',
  templateUrl: 'recents.component.html',
  styleUrls: ['./recents.component.css']
})

@Injectable()
export class RecentTacosComponent implements OnInit {
  recentTacos: any;

  constructor(private httpClient: HttpClient) { }

  ngOnInit() {
    this.httpClient.get('http://localhost:8080/design/recent') ←--- 从服务器端获取最近的taco
        .subscribe(data => this.recentTacos = data);
  }
}
```

我们需要关注ngOnInit()方法。在这个方法中，RecentTacosComponent使用注入的Http模块来针对http://localhost:8080/design/recent地址发送HTTP GET请求，并期望得到一个包含taco设计的列表，它们会被放到名为recentTacos的模型属性中。视图（在recents.component.html中）会将模型数据展现为HTML的形式，以便于在浏览器中渲染。在创建完3个taco之后，最终的结果如图6.2所示。

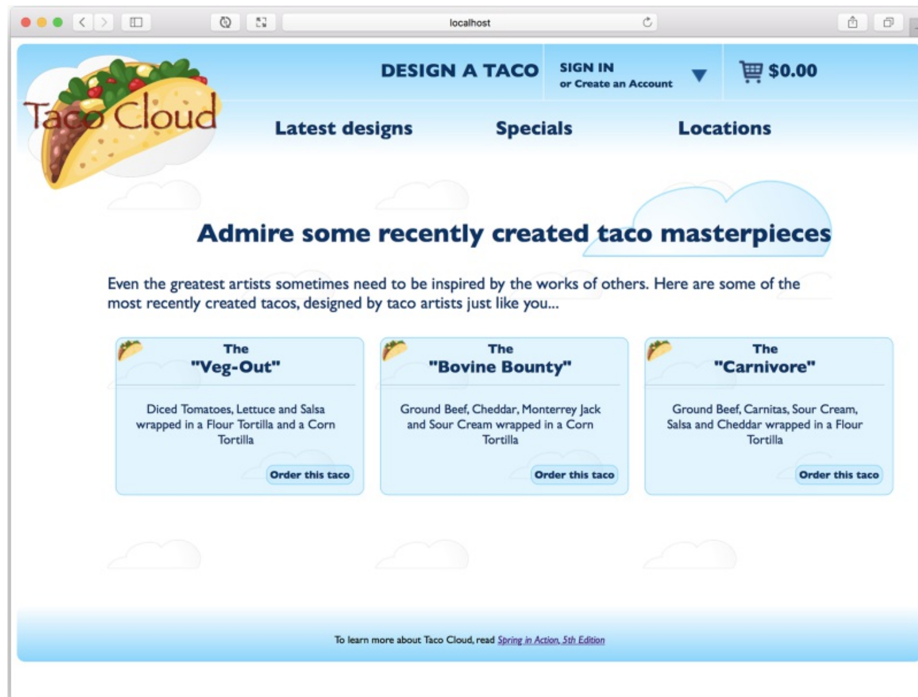


图6.2　展现最近创建的taco

在我们的拼图中，缺失的一部分就是端点，它会处理针对“/design/recent”的HTTP GET请求并将最近设计的taco列表作为响应。我们需要创建一个新的控制器来处理这种请求。程序清单6.2展现了完成该任务的控制器。

程序清单6.2　处理taco设计API请求的RESTful控制器

```
package tacos.web.api;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.hateoas.EntityLinks;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;
import tacos.data.TacoRepository;

@RestController
@RequestMapping(path="/design",               ←--- 处理针对“/design”的请求
                produces="application/json")
@CrossOrigin(origins="*")                         ←--- 允许跨域请求
public class DesignTacoController {
  private TacoRepository tacoRepo;

  @Autowired
  EntityLinks entityLinks;

  public DesignTacoController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping("/recent")
  public Iterable<Taco> recentTacos() {           ←--- 获取并返回最近设计的
taco
    PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
    return tacoRepo.findAll(page).getContent();
  }
}
```

你可能会觉得这个控制器的名字看起来非常熟悉。在第2章中，我们创建了名为DesignTacoController的控制器，它会处理类似的请求类型。但是，当时是用来处理多页Taco Cloud应用的，这个新的DesignTacoController是一个REST控制器，是由@RestController注解声明的。

@RestController注解有两个目的。首先，它是一个类似于@Controller和@Service的构造型注解，能够让类被组件扫描功能发现。但是，与REST最密切相关之处在于，@RestController注解会告诉

Spring，控制器中的所有处理器方法的返回值都要直接写入响应体中，而不是将值放到模型中并传递给一个视图以便于进行渲染。

作为替代方案，我们也可以像其他Spring MVC控制器那样为DesignTacoController添加@Controller注解。但是，这样的话，我们就需要为每个处理器方法再添加@ResponseBody注解，这样才能达到相同的效果。另外一种方案就是返回ResponseEntity对象，我们稍后将会对其进行讨论。

类级别的@RequestMapping注解，再加上recentTacos()方法上的@GetMapping注解，两者结合起来指定recentTacos()方法将会负责处理针对“/design/recent”的GET请求（这也正是Angular代码所需要的）。

你还会发现，@RequestMapping注解还设置了一个 produces 属性。这指明DesignTacoController中的所有处理器方法只会处理Accept头信息包含“application/json”的请求。它不仅会限制API只会生成JSON结果，同时还允许其他的控制器（比如第2章中的DesignTacoController）处理具有相同路径的请求，只要这些请求不要求JSON格式的输出就可以。尽管这样会限制API是基于JSON的，但是我们还可以将produces设置为一个String类型的数组，这样的话就允许我们设置多个内容类型。比如，为了允许生成XML格式的输出，我们可以为produces属性添加“text/xml”：

```
@RequestMapping(path="/design",
                produces={"application/json", "text/xml"})
```

在程序清单6.2中，你可能还发现这个类添加了@CrossOrigin注解。因为应用程序的Angular部分将会运行在与API相独立的主机和/或端口上（至少目前是这样的），Web浏览器会阻止Angular客户端消费该API。我们可以在服务端响应中添加CORS（Cross-Origin Resource Sharing，跨域资源共享）头信息来突破这一限制。Spring借助@CrossOrigin注解让CORS的使用更加简单。正如我们所看到的，@CrossOrigin允许来自任何域的客户端消费该API。

recentTacos()方法中的逻辑非常简单直接。它构建了一个PageRequest对象，指明我们想要第一页（序号为0）的12条结果，并且要按照taco的创建时间降序排列。简而言之，我们想要得到12个最近创建的taco设计。PageRequest会被传递到TacoRepository的findAll()方法中，分页的结果内容则会返回到客户端（也就是在程序清单6.2中我们所看到的，它们将会以模型数据展现给用户）。

现在，假设我们想要提供一个按照ID抓取单个taco的端点。通过在处理器方法的路径上使用占位符并让方法接收一个路径变量，我们能够捕获到这个ID，然后就可以借助repository查找Taco对象了：

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return optTaco.get();
  }
  return null;
}
```

因为控制器的基础路径是“/design”，所以这个控制器方法处理的是

针对“/design/{id}”的GET请求，其中路径的“{id}”部分是占位符。请求中的实际值将会传递给id参数，它通过@PathVariable注解与{id}占位符进行匹配。

在tacoById()中，id参数被传递到了repository的findById()方法中，以便于抓取Taco。findById()返回的是一个Optional<Taco>，因为根据给定的ID可能获取不到taco，所以在返回值的时候我们需要确定该ID是否能够匹配一个taco。如果能够匹配，我们可以调用Optional<Taco>对象的get()方法返回实际的Taco。

如果该ID无法匹配任何已知的taco，我们将会返回null。但是，这种做法并不完美。如果我们返回null，客户端将会接收到空的响应体以及值为200(OK)的HTTP状态码。客户端实际上接收到了一个无法使用的响应，但是状态码却提示一切正常。有一种更好的方式是在响应中使用HTTP 404 (NOT FOUND)状态。

按照现在的写法，我们没有简单的途径在tacoById()中返回404状态。但是，如果我们做一些小的调整，就可以将状态码设置成很恰当的值了：

```
@GetMapping("/{id}")
public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
  }
  return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
}
```

现在，tacoById()返回的不是一个Taco对象，而是ResponseEntity<Taco>。如果找到taco，我们就将Taco包装到ResponseEntity中，并且会带有OK的HTTP状态（这也是之前的行为）。如果找不到taco，我们就将会在ResponseEntity中包装一个null，并且会带有NOT FOUND的HTTP状态，从而表明客户端试图抓取的taco并不存在。

我们已经有了面向Angular客户端的初始Taco Cloud API，当然它也可以用于其他类型的客户端。在开发中，我们可能还想使用像curl或HTTPie这样的命令行工具来探测该API。比如，如下的命令行展示了如何通过curl获取最新创建的taco：

```
$ curl localhost:8080/design/recent
```

如果你更喜欢HTTPie，那如下所示：

```
$ http :8080/design/recent
```

定义能够返回信息的端点仅仅是第一步。如果我们的API需要从客户端接收数据又该怎么办呢？接下来，我们看一下如何编写控制器来处理请求的输入。

6.1.2　发送数据到服务器端

到目前为止，我们的API能够返回多个最近创建的taco。但是，这些taco又是怎样创建的呢？

我们还没有删掉第2章的任何代码，所以原始的DesignTacoController还存在，它会展现taco的设计表单并处理表单的提交。这是获取测试数据来验证我们所创建的API的一个好办法。如果我们想要将Taco Cloud转换成单页应用，那么我们需要创建Angular组件以及对应的端点，以此来替换第2章中的taco设计表单。

在客户端代码方面，我们通过一个名为DesignComponent（在名为design.component.ts的文件中）的新Angular组件来处理taco设计表单。因为要处理表单提交，所以DesignComponent中有一个onSubmit()方法，如下所示：

```
onSubmit() {
  this.httpClient.post(
      'http://localhost:8080/design',
      this.model, {
          headers: new HttpHeaders().set('Content-type', 'application/json
'),
      }).subscribe(taco => this.cart.addToCart(taco));

  this.router.navigate(['/cart']);
}
```

在onSubmit()方法中，我们调用了HttpClient的post()方法而不是get()方法。这意味着我们不再是从API中抓取数据，而是向API发送数据。具体来讲，我们将一个taco设计（存放到model变量中）借助HTTP POST请求发送至“/design”的API端点上。

因此，我们需要在DesignTacoController中编写一个方法处理该请求并保存该taco设计。通过在DesignTacoController中添加如下的postTaco()方法，我们就能让控制器实现该功能：

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
  return tacoRepo.save(taco);
}
```

因为postTaco()将会处理HTTP POST请求，所以它使用了@PostMapping注解，而不是@GetMapping。在这里，我们没有指定path属性，因此按照DesignTacoController上的类级别@RequestMapping注解，postTaco()方法将会处理对“/design”的请求。

但是，我们设置了consumes属性。consumes属性用于指定请求输入，而produces用于指定请求输出。在这里，我们使用consumes属性，表明该方法只会处理Content-type与application/json相匹配的请求。

方法的Taco参数带有@RequestBody注解，表明请求应该被转换为一个Taco对象并绑定到该参数上。这个注解是非常重要的，如果没有它，Spring MVC将会认为我们希望将请求参数（要么是查询参数，要么是表单参数）绑定到Taco上。但是，@RequestBody注解能够确保请求体中的JSON会被绑定到Taco对象上。

在postTaco()接收到Taco对象之后，就会将该对象传递给TacoRepository的save()方法。

你可能也注意到了，我为postTaco()方法添加了@ResponseStatus(HttpStatus.CREATED)注解。在正常的情况下（没有异常抛出的时候），所有响应的HTTP状态码都是200 (OK)，表明请求是成功的。尽管我们始终都希望得到HTTP 200，但是有些时候它的描述

性不足。在POST请求的情况下，201 (CREATED)的HTTP状态更具有描述性。它会告诉客户端，请求不仅成功了，还创建了一个资源。在适当的地方使用@ResponseStatus将最具描述性和最精确的HTTP状态码传递给客户端是一种更好的做法。

我们已经使用@PostMapping创建了新的Taco资源，除此之外，POST请求还能用来更新资源。尽管如此，POST请求通常用来创建资源，而PUT和PATCH请求通常用来更新资源。接下来，让我们看一下该如何使用@PutMapping和@PatchMapping来更新数据。

6.1.3 在服务器上更新数据

在编写控制器来处理HTTP PUT或PATCH命令之前，我们应该花点时间直面这个问题：为什么会有两种不同的HTTP方法来更新资源？

尽管PUT经常被用来更新资源，但它在语义上其实是GET的对立面。GET请求用来从服务端往客户端传输数据，而PUT请求则是从客户端往服务端发送数据。

从这个意义上讲，PUT真正的目的是执行大规模的替换（replacement）操作，而不是更新操作。HTTP PATCH的目的是对资源数据打补丁或局部更新。

例如，假设我们想要更新某个订单的地址信息。借助REST API，其中有一种实现方式就是借助如下所示的PUT请求处理器：

```
@PutMapping("/{orderId}")
public Order putOrder(@RequestBody Order order) {
  return repo.save(order);
}
```

这种方式可以运行，但是它可能需要客户端将完整的订单数据从PUT请求中提交上来。从语义上讲，PUT意味着“将数据放到这个URL上”，其本质上就是替换已有的数据。如果省略了订单上的某个属性，那么该属性的值应该被null所覆盖，甚至订单中的taco也需要和订单数据一起设置，否则，它们将会从订单中移除。

如果PUT请求所做的是对资源数据的大规模替换，那么我们该如何处理局部更新的请求呢？这就是HTTP PATCH请求和Spring的@PatchMapping注解所擅长的事情了。如下展示了如何编写控制器方法来处理订单的PATCH请求：

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public Order patchOrder(@PathVariable("orderId") Long orderId,
                        @RequestBody Order patch) {

  Order order = repo.findById(orderId).get();
  if (patch.getDeliveryName() != null) {
    order.setDeliveryName(patch.getDeliveryName());
  }
  if (patch.getDeliveryStreet() != null) {
    order.setDeliveryStreet(patch.getDeliveryStreet());
  }
  if (patch.getDeliveryCity() != null) {
    order.setDeliveryCity(patch.getDeliveryCity());
  }
  if (patch.getDeliveryState() != null) {
    order.setDeliveryState(patch.getDeliveryState());
  }
  if (patch.getDeliveryZip() != null) {
    order.setDeliveryZip(patch.getDeliveryState());
  }
  if (patch.getCcNumber() != null) {
```

```
    order.setCcNumber(patch.getCcNumber());
  }
  if (patch.getCcExpiration() != null) {
    order.setCcExpiration(patch.getCcExpiration());
  }
  if (patch.getCcCVV() != null) {
    order.setCcCVV(patch.getCcCVV());
  }

  return repo.save(order);
}
```

这里需要关注的第一件事情就是patchOrder()方法使用了@PatchMapping注解，而不是@PutMapping注解，这表示它应该处理HTTP PATCH请求，而不是PUT请求。

有一点你肯定也注意到了，那就是patchOrder()方法比putOrder()方法要更复杂一些。这是因为Spring MVC的映射注解，虽然包括了@PatchMapping和@PutMapping，但是它们只能用来指定某个方法能够处理什么类型的请求，这些注解并没有规定该如何处理请求，尽管PATCH在语义上代表局部更新，但是在处理器方法中实际编写代码执行更新的还是我们自己。

对于putOrder()方法来说，我们得到的是完整的订单数据，然后将它保存起来，这样就完全符合HTTP PUT的语义。但是，对于patchMapping()来说，为了符合HTTP PATCH的语义，方法体需要更多的智慧才行。在这里，我们不是用新发送过来的数据完全替换已有的订单，而是探查传入Order对象的每个字段，并将所有非null的值应用到已有的订单上。这种方式允许客户端只发送要改变的属性就可以，并且对于客户端没有指定的属性，服务器端会保留已有的数据。

还有更多的方式来实现**PATCH**

patchOrder()方法中的PATCH操作还有一些限制：

- 如果null意味着没有变化，那么客户端该如何指定一个字段真的要设置为null？
- 我们没有办法移除或添加集合的子集。如果客户端想要添加或移除集合中的条目，那么它必须将变更的完整集合发送到服务器端。

关于PATCH请求该如何处理以及传入的数据该是什么样子并没有硬性的规定。客户端可以发送一个PATCH请求特定的变更描述，而不是发送真正的领域数据。当然，如果是这样，那么请求处理器方法就会改写为处理patch指令，而不是领域数据。

在@PutMapping和@PatchMapping中，需要注意引用的请求路径都是要进行变更的资源。这与@GetMapping注解标注的方法在处理路径时的方式是相同的。

我们已经看过了如何使用@GetMapping和@PostMapping获取和发送资源。同时，也看到了使用@PutMapping和@PatchMapping更新资源的两种方式。剩下的就是该如何处理删除资源的请求了。

6.1.4　删除服务器上的数据

有时，有些数据可能不再需要了。在这种场景下，客户端应该能够通过HTTP DELETE请求的形式要求移除某个资源。

Spring MVC的@DeleteMapping注解能够非常便利地声明处理DELETE请求的方法。例如，我们想要有一个能够删除订单资源的API。如下的控制器方法就能做到这一点：

```
@DeleteMapping("/{orderId}")
@ResponseStatus(code=HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId) {
  try {
    repo.deleteById(orderId);
  } catch (EmptyResultDataAccessException e) {}
}
```

现在，再向你解释这个映射注解就有些啰唆了。我们已经见过了@GetMapping、@PostMapping、@PutMapping和@PatchMapping，每个注解都能够指定某个方法可以处理对应类型的HTTP请求。毫无疑问，@DeleteMapping会指定deleteOrder()方法负责处理针对“/orders/{orderId}”的DELETE请求。

这个方法中的代码会负责真正删除订单。在本例中，它会接收订单ID并将其传递给repository的deleteById()方法，其中这个ID是以URL中路径变量的形式提供的。如果方法调用的时候该订单存在，就将会删除这个订单。如果订单不存在，就会抛出EmptyResultDataAccessException。

在这里，我选择捕获该EmptyResultDataAccessException异常，但是什么都没有做。在这里，我的想法是如果你尝试删除一个并不存在的资源，那么它的结果和删除之前存在这个资源是一样的。也就是，最终的效果都是资源不复存在。所以在删除之前资源是否存在并不重要。另外一种办法就是可以让deleteOrder()返回ResponseEntity，在资源不存在的时候将响应体设置为null并将HTTP状态码设置为NOT FOUND。

deleteOrder()方法唯一需要注意的是它使用了@ResponseStatus注解，以确保响应的HTTP状态码为204（NO CONTENT）。对于已经不存在的资源，我们没有必要返回任何的资源数据给客户端，因此DELETE请求通常并没有响应体，我们需要以HTTP状态码的形式让客户端知道不要期望得到任何的内容。

现在，Taco Cloud API已经基本成形了。客户端的代码可以很容易地消费我们的API，以便于显示配料、接收订单和展示最近创建的taco。但是，我们还可以更进一步，让API更易于客户端消费。接下来，我们看一下如何为Taco Cloud API添加超媒体功能。

6.2 启用超媒体

到目前为止，我们所创建的API非常简单，但是只要消费它的客户端知道API的URL模式，它们就可以正常运行。例如，客户端可能会以硬编码的形式对“/design/recent”发送GET请求，以便于获取最近创建的taco。类似的，客户端会以硬编码的形式将taco列表中的ID拼接到“/design”上形成获取特定taco资源的URL。

在API客户端编码中，使用硬编码模式和字符串操作是很常见的。但是，我们设想一下，如果API的URL模式发生了变化又会怎么样呢？硬编码的客户端代码掌握的依然是旧的API信息，因此客户端代码将无法正常运行。对API URL进行硬编码和字符串操作会让客户端代码变得很脆弱。

超媒体作为应用状态引擎（Hypermedia as the Engine of Application State，HATEOAS）是一种创建自描述API的方式。API所返回的资源中会包含相关资源的链接，客户端只需要了解最少的API URL信息就能导航整个API。这种方式能够掌握API所提供的资源之间的关系，客户端能够基于API的URL中所发现的关系对它们进行遍历。

举例来说，假设某个客户端想要请求最近设计的taco的列表，按照原始的形式，在没有超链接的情况下，客户端以JSON格式接收到的taco列表会如下所示（为了简洁，这里只保留了第一个taco，剩余的省略了）：

```
[
  {
    "id": 4,
    "name": "Veg-Out",
    "createdAt": "2018-01-31T20:15:53.219+0000",
    "ingredients": [
      {"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},
      {"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},
      {"id": "TMTO", "name": "Diced Tomatoes", "type": "VEGGIES"},
      {"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},
      {"id": "SLSA", "name": "Salsa", "type": "SAUCE"}
    ]
  },
  ...
]
```

如果客户端想要获取某个taco或者对其进行其他HTTP操作，就需要将它的id属性以硬编码的方式拼接到一个路径为“/design”的URL上。与之类似，如果客户端想要对某个配料执行HTTP请求，就需要将该配料id属性的值拼接到路径为“/ingredients”的URL上。在这两种情况下，都需要在路径上添加“http://”或“https://”前缀以及API的主机名。

如果API启用了超媒体功能，那么API将会描述自己的URL，从而减轻客户端对其进行硬编码的痛苦。如果嵌入超链接，那么最近创建的taco列表将会如程序清单6.3所示。

程序清单6.3　包含超链接的taco资源列表

```
{
  "_embedded": {
    "tacoResourceList": [
      {
        "name": "Veg-Out",
        "createdAt": "2018-01-31T20:15:53.219+0000",
        "ingredients": [
          {
            "name": "Flour Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/FLTO" }
            }
          },
          {
            "name": "Corn Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/COTO" }
            }
          },
          {
            "name": "Diced Tomatoes", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/TMTO" }
            }
          },
          {
```

```
            "name": "Lettuce", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/LETC" }
            }
          },
          {
            "name": "Salsa", "type": "SAUCE",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/SLSA" }
            }
          }
        ],
        "_links": {
          "self": { "href": "http://localhost:8080/design/4" }
        }
      },

      ...
    ]
  },
  "_links": {
    "recents": {
      "href": "http://localhost:8080/design/recent"
    }
  }
}
```

这种特殊风格的HATEOAS被称为HAL（超文本应用语言，Hypertext Application Language）。这是一种在JSON响应中嵌入超链接的简单通用格式。

虽然这个列表看上去不像前面那样简洁，但是它确实提供了一些有用的信息。这个新taco列表中的每个元素都包含了一个名为“_links”的属性，为客户端提供导航API的超链接。在本例中，taco和配料都有一个“self”链接，用来引用该资源；整个列表有一个“recents”链接，用来引用该API自身。

如果客户端应用需要对列表中的taco执行HTTP请求，那么在开发的

时候不需要关心taco资源的URL是什么样子。相反，它只需要请求“self”链接就可以了，该属性将会映射至http://localhost:8080/design/4。如果客户端想要处理特定的配料，只需要查找该配料的“self”链接即可。

Spring HATEOAS项目为Spring提供了超链接的支持。它提供了一些类和资源装配器（assembler），在Spring MVC控制器返回资源之前能够为其添加链接。

为了在Taco Cloud API中启用超媒体功能，我们需要在构建文件中添加如下的Spring HATEOAS starter依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

这个starter不仅会将Spring HATEOAS添加到项目的类路径中，还会提供自动配置功能以启用Spring HATEOAS。我们所需要做的就是重新实现控制器，让它们返回资源类型，而不是领域类型。

我们首先为最近taco列表添加超链接，也就是针对“/design/recent”的GET请求。

6.2.1　添加超链接

Spring HATEOAS提供了两个主要的类型来表示超链接资源：

Resource和Resources。Resource代表一个资源，而Resources代表资源的集合。这两种类型都能携带到其他资源的链接。当从Spring MVC REST控制器返回时，它们所携带的链接将会包含到客户端所接收到的JSON（或XML）中。

为了给最近创建的taco添加超链接，我们需要重新实现程序清单6.2中的recentTacos()方法。原始的实现返回的是List<Taco>，当时这种返回值是可以的，但是现在我们需要让它返回Resources对象。程序清单6.4展示了recentTacos()的新实现，包含了在最近taco列表中启用超链接的第一步。

程序清单6.4　为资源添加超链接

```
@GetMapping("/recent")
public Resources<Resource<Taco>> recentTacos() {
  PageRequest page = PageRequest.of(
          0, 12, Sort.by("createdAt").descending());

  List<Taco> tacos = tacoRepo.findAll(page).getContent();
  Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

  recentResources.add(
      new Link("http://localhost:8080/design/recent", "recents"));
  return recentResources;
}
```

在这个新版本的recentTacos()中，我们不再直接返回taco的列表，而是使用Resources.wrap()将taco列表包装为Resources<Resource<Taco>>，并使其作为该方法最终的返回值。但是在Resources对象返回之前，我们添加了名为recents的关联关系，它的URL为http://localhost:8080/design/recent。这样做的结果就是，API请求所返回

的资源中将会包含如下的JSON片段：

```
"_links": {
  "recents": {
    "href": "http://localhost:8080/design/recent"
  }
}
```

这是一个很好的起点，但是我们还有一些事情需要完成。现在，我们只是为整体的列表添加了链接，还没有为taco资源本身以及每个taco中的配料添加链接。我们很快就会实现该功能，但是在此之前，我们要先解决recents链接中的硬编码问题。

像这样对URL进行硬编码是一种很糟糕的办法。除非Taco Cloud的目标仅限于在本地开发机器上运行应用，否则，我们需要找一种方式避免在URL中使用硬编码的localhost:8080。幸运的是，Spring HATEOAS以链接构建者（link builder）的方式为我们提供了帮助。

在Spring HATEOAS中，最有用的链接构建者是ControllerLinkBuilder。这个链接构建者非常智能，它能自动探知主机名是什么，这样就能避免对其进行硬编码。同时，它还提供了流畅的API，允许我们相对于控制器的基础URL构建链接。

借助ControllerLinkBuilder，我们可以将recentTacos()中硬编码的Link创建改造成如下的形式：

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);
recentResources.add(
  ControllerLinkBuilder.linkTo(DesignTacoController.class)
                       .slash("recent")
                       .withRel("recents"));
```

我们不仅不需要硬编码主机名，而且不再需要指定“/design”。在这里，我们向DesignTacoController请求获取一个链接，它的基础路径为“/design”。ControllerLinkBuilder使用控制器的基础路径作为我们创建的Link对象的基础。

接下来，调用了在Spring项目中我最喜欢的一个方法：slash()。我喜欢这个方法的原因是这个方法非常简洁地描述了它要做的事情。这个方法会为URL添加斜线（/）和给定的值，所形成的URL路径是“/design/recent”。

最后，我们为Link指定了一个关系名。在本例中，关系名为recents。

尽管我是slash()的忠实粉丝，但是ControllerLinkBuilder还有另外一个方法，能够消除链接URL上的所有硬编码。此时，我们不再需要调用slash()，而是调用linkTo()，并将控制器中的一个方法传递给它，这样ControllerLinkBuilder就能推断出控制器的基础路径和该方法的映射路径。如下的代码就以这种方式使用了linkTo()方法：

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);
recentResources.add(
        linkTo(methodOn(DesignTacoController.class).recentTacos())
        .withRel("recents"));
```

在这里，我静态导入了linkTo()和methodOn()方法（它们都来自ControllerLinkBuilder），从而让代码更易于阅读。methodOn()方法传入控制器类，从而允许我们调用recentTacos()方法，这个调用会被

ControllerLinkBuilder拦截，用来确定控制器的基础路径和recentTacos()的映射路径。现在，整个URL都从控制器的映射中判断出来了，而且完全没有硬编码。非常棒！

6.2.2 创建资源装配器

现在，我们需要为列表中的taco资源添加链接。有种方案就是遍历Resources对象中所携带的每个Resource<Taco>元素，为它们依次添加Link。但是，这种方式有点过于枯燥，在需要返回taco列表的所有地方都需要在API中重复循环相关的代码。

我们需要有一种不同的策略。

对于列表中的每个taco，我们不再使用Resources.wrap()来创建Resource，而是定义一个将Taco对象转换为TacoResource对象的工具类。TacoResource对象与Taco类似，但是它本身能携带链接。程序清单6.5展示了TacoResource。

程序清单6.5　能够携带领域数据和超链接列表taco资源

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

  @Getter
```

```
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
  private final List<Ingredient> ingredients;

  public TacoResource(Taco taco) {
    this.name = taco.getName();
    this.createdAt = taco.getCreatedAt();
    this.ingredients = taco.getIngredients();
  }

}
```

从很多方面来看，TacoResource都与Taco领域类型没有区别。它们都有name、createdAt和ingredients属性。但是，TacoResource扩展了ResourceSupport，从而继承了一个Link对象的列表和管理链接列表的方法。

除此之外，TacoResource并没有包含Taco的id属性。这是因为没有必要在API中暴露数据库相关的ID。从API客户端的角度来看，资源的self链接将会作为该资源的标识符。

注意："领域"和"资源"应该各自独立还是应该是同一个类呢？有些Spring开发人员可能会让领域类型扩展ResourceSupport，从而将领域和资源对象合二为一。至于哪种方式才是正确的，这里并没有确切答案。我选择的做法是创建一个单独的资源类型，这样Taco就没有必要添加资源链接了，因为在有些场景下，这些链接是根本用不到的。另外，通过创建一个单

独的资源类型，能够很容易地将id属性排除出去，这样它就不会暴露在API中了。

TacoResource有一个很简单的构造器，会接收一个Taco对象并且会将Taco中的相关属性复制到自己的属性中。这样的话，我们可以很容易地将一个Taco对象转换为TacoResource。但是，如果我们就此止步，就依然需要遍历Taco列表才能将其转换成Resources<TacoResource>。

为了将Taco对象转换成TacoResource对象，我们需要创建一个资源装配器（resource assembler）。我们所需要的装配器如程序清单6.6所示。

程序清单6.6　装配taco资源的资源装配器

```
package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Taco;

public class TacoResourceAssembler
       extends ResourceAssemblerSupport<Taco, TacoResource> {

  public TacoResourceAssembler() {
    super(DesignTacoController.class, TacoResource.class);
  }

  @Override
  protected TacoResource instantiateResource(Taco taco) {
    return new TacoResource(taco);
  }

  @Override
  public TacoResource toResource(Taco taco) {
    return createResourceWithId(taco.getId(), taco);
```

```
  }
}
```

TacoResourceAssembler有一个默认的构造器，会告诉超类（ResourceAssemblerSupport）在创建TacoResource中的链接时将会使用DesignTacoController来确定所有URL的基础路径。

instantiateResource()方法进行了重写，以便基于给定的Taco实例化TacoResource。如果TacoResource有默认构造器，那么这个方法是可选的。但是，在本例中，TacoResource的构造过程需要Taco，所以我们要重写它。

最后是toResource()方法，这是在扩展ResourceAssemblerSupport时唯一强制实现的方法。在这里，我们告诉它要通过Taco创建TacoResource，并且要设置一个self链接，这个链接的URL是根据Taco对象的id属性衍生出来的。

在表面上，toResource()和instantiateResource()的用途很相似，但是它们的目的略有不同。instantiateResource()只是为了实例化一个Resource对象，而toResource()的意图不仅是创建Resource对象，还要为其填充链接。在内部，toResource()将会调用instantiateResource()。

现在，我们调整一下recentTacos()，让它使用TacoResourceAssembler：

```
@GetMapping("/recent")
public Resources<TacoResource> recentTacos() {
  PageRequest page = PageRequest.of(
```

```
        0, 12, Sort.by("createdAt").descending());
  List<Taco> tacos = tacoRepo.findAll(page).getContent();

  List<TacoResource> tacoResources =
      new TacoResourceAssembler().toResources(tacos);
  Resources<TacoResource> recentResources =
      new Resources<TacoResource>(tacoResources);
  recentResources.add(
      linkTo(methodOn(DesignTacoController.class).recentTacos())
      .withRel("recents"));
  return recentResources;
}
```

在这里，recentTacos()的返回值不再是Resources<Resource<Taco>>类型，而是利用我们新定义的TacoResource类型返回了一个Resources<TacoResource>。在从repository获取taco之后，我们将Taco对象的列表传递给TacoResourceAssembler的toResources()方法。这个便利的方法会循环所有的Taco对象，调用我们在TacoResourceAssembler中重写的toResource()方法来创建TacoResource对象的列表。

在有了TacoResource列表之后，我们接下来创建了Resources<TacoResource>对象，然后像前面版本的recentTacos()一样，为其填充了recents链接。

此时，对“/design/recent”发起GET请求将会生成taco的一个列表，其中的每个taco都有一个self链接，而列表整体有一个recents链接。但是，配料目前还没有链接。为了解决这个问题，我们需要为配料创建一个新的资源装配器：

```
package tacos.web.api;
import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import tacos.Ingredient;
```

```
class IngredientResourceAssembler extends
          ResourceAssemblerSupport<Ingredient, IngredientResource> {

  public IngredientResourceAssembler() {
    super(IngredientController2.class, IngredientResource.class);
  }

  @Override
  public IngredientResource toResource(Ingredient ingredient) {
    return createResourceWithId(ingredient.getId(), ingredient);
  }

  @Override
  protected IngredientResource instantiateResource(
                                            Ingredient ingredient) {
    return new IngredientResource(ingredient);
  }

}
```

我们可以看到，IngredientResourceAssembler与TacoResourceAssembler非常相似，但是它使用的是Ingredient和IngredientResource对象而不是Taco和TacoResource对象。

谈到IngredientResource对象，它的源码如下所示：

```
package tacos.web.api;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Ingredient.Type;

public class IngredientResource extends ResourceSupport {

  @Getter
  private String name;

  @Getter
  private Type type;

  public IngredientResource(Ingredient ingredient) {
    this.name = ingredient.getName();
    this.type = ingredient.getType();
```

```
  }
}
```

与TacoResource类似，IngredientResource扩展了ResourceSupport，并且会将领域类型相关的属性复制到自己的属性中（id属性除外）。

接下来，我们需要修改一下TacoResource，让它能够携带IngredientResource对象，而不是Ingredient对象：

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

  private static final IngredientResourceAssembler
            ingredientAssembler = new IngredientResourceAssembler();

  @Getter
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
  private final List<IngredientResource> ingredients;

  public TacoResource(Taco taco) {
    this.name = taco.getName();
    this.createdAt = taco.getCreatedAt();
    this.ingredients =
        ingredientAssembler.toResources(taco.getIngredients());
  }

}
```

这个新版本的TacoResource会创建一个static final的

IngredientResourceAssembler实例，并且会使用它的toResource()方法将给定Taco对象的Ingredient列表转换成IngredientResource列表。

我们现在的taco列表已经完全具备了超链接，不仅是它本身（recents链接），而且所有的taco条目和每个taco中的配料都有了超链接。响应的内容应该与程序清单6.3非常相似了。

你可以就此止步并跳到下一节，但是在此之前，我想解决程序清单6.3中一些不太理想的问题。

6.2.3 命名嵌套式的关联关系

如果你仔细看一下程序清单6.3，就会发现顶层的元素如下所示：

```
{
  "_embedded": {
    "tacoResourceList": [
      ...
    ]
  }
}
```

最值得注意的是在embedded之下有一个名为tacoResourceList的属性。之所以有这个名称，是因为Resources对象是通过List<TacoResource>创建出来的。尽管可能性不太大，但是假设我们将TacoResource类重构成了其他的名称，那么结果JSON中的字段名将会随之发生变化。这样，所有依赖该名称的客户端代码都会产生问题。

@Relation注解能够帮助我们消除JSON字段名和Java代码中定义的

资源类名之间的耦合。通过为TacoResource添加@Relation注解，我们就能指定Spring HATEOAS该如何命名结果JSON中的字段名：

```
@Relation(value="taco", collectionRelation="tacos")
public class TacoResource extends ResourceSupport {
  ...
}
```

在这里，我们指定当在Resources对象中引用TacoResource对象列表时它应该被命名为tacos。虽然在我们的API中没有用到，但是如果在JSON中引用单个TacoResource对象，那么它的名字将会是taco。这样的话，"/design/recent"所返回的JSON将会如下所示（不管我们是否要对TacoResource进行重构，这个结构都不会发生变化）：

```
{
  "_embedded": {
    "tacos": [
      ...
    ]
  }
}
```

借助Spring HATEOAS，向API中添加链接变得非常简单直接。尽管如此，它也会添加一些额外的代码。所以，很多开发人员会选择在API中不使用HATEOAS，但是如果API的URL模式发生变化，那么客户端代码就不可用了。所以，我建议你认真考虑一下HATEOAS，不要因为偷懒而忽略在资源中添加超链接。

如果你真的想要偷懒，那么只要你使用Spring Data来实现repository，我们就还有一个双赢的方案。接下来，我们看一下基于在第3章中使用Spring Data所创建的数据repository，如何借助Spring Data

REST自动创建API。

6.3 启用数据后端服务

正如我们在第3章中所看到的，Spring Data有一种特殊的魔法，它能够基于我们定义的接口自动创建repository实现。但是Spring Data还有另外一项技巧，它能帮助我们定义应用的API。

Spring Data REST是Spring Data家族中的另外一个成员，它会为Spring Data创建的repository自动生成REST API。我们只需要将Spring Data REST添加到构建文件中，就能得到一套API，它的操作与我们定义repository接口是一致的。

为了使用Spring Data REST，我们需要将如下的依赖添加到构建文件中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

不管你是否相信，对于已经使用Spring Data自动生成repository的项目来说，我们只需要完成这一步就能对外暴露REST API了。将Spring Data REST starter添加到构建文件中之后，应用的自动配置功能会为Spring Data（包括Spring Data JPA、Spring Data Mongo等）创建的所有repository自动创建REST API。

Spring Data REST所创建的端点和我们自己创建的端点一样好（甚

至能够更好一些）。所以，我们可以做一些移除操作，在进行下一步之前将我们已经创建的带有@RestController注解的类移除。

为了尝试Spring Data REST提供的端点，我们可以启动应用并测试一些URL。基于为Taco Cloud定义的repository，我们可以对taco、配料、订单和用户执行一些GET请求。

举例来说，我们可以向“/ingredients”发送GET请求以获取所有配料。借助curl，我们得到的响应大致如下所示（进行了删减，只显示第一种配料）：

```
$ curl localhost:8080/ingredients
{
  "_embedded" : {
    "ingredients" : [ {
      "name" : "Flour Tortilla",
      "type" : "WRAP",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        },
        "ingredient" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        }
      }
    },
    ...
    ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/ingredients"
    }
  }
}
```

太棒了！我们只不过将一项依赖添加到了构建文件中，这样不但得到了针对配料的端点，而且返回的资源中还包含超链接。我们可以假装成这个API的客户端，使用curl继续访问self链接以获取玉米面薄饼的详情：

```
$ curl http://localhost:8080/ingredients/FLTO
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    }
  }
}
```

为了避免分散注意力，在本书中，我们不再浪费时间深入探究Spring Data REST所创建的每个端点和可选项。但是，我们需要知道，它还支持端点的POST、PUT和DELETE方法。也就是说，你可以发送POST请求至“/ingredients”来创建新的配料，也可以发送DELETE请求到“/ingredients/FLTO”，以便于从菜单中删除玉米面薄饼。

我们想做的另外一件事可能就是为API设置一个基础路径，这样它们会有不同的端点，避免与我们所编写的控制器产生冲突（实际上，如果我们不删除前面自己创建的IngredientsController，就将会干扰Spring Data REST提供的“/ingredients”端点）。为了调整API的基础路径，我们可以设置spring.data.rest.base-path属性：

```
spring:
```

```
  data:
    rest:
      base-path: /api
```

这项配置会将Spring Data REST端点的基础路径设置为“/api”。现在，配料端点将会变成“/api/ingredients”。我们通过请求taco列表来验证一下这个新的基础路径：

```
$ curl http://localhost:8080/api/tacos
{
  "timestamp": "2018-02-11T16:22:12.381+0000",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/api/tacos"
}
```

哦，天啊！它并没有按照预期那样运行。我们有了Ingredient实体和IngredientRepository接口之后，Spring Data REST就会暴露“/api/ingredients”。我们也有Taco实体和TacoRepository接口，为什么Spring Data REST没有为我们生成“/api/tacos”端点呢？

6.3.1　调整资源路径和关系名称

实际上，Spring Data确实为我们提供了处理taco的端点。虽然Spring Data REST非常聪明，但是在暴露taco端点的时候出现了一点问题。

当为Spring Data repository创建端点的时候，Spring Data REST会尝试使用相关实体类的复数形式。对于Ingredient实体来说，端点将会是“/ingredients”，对于Order和User实体，端点将会

是“/orders”和“/users”。到目前为止，一切运行良好。但有些场景下，比如遇到“taco”的时候，它获取到这个单词，为其生成的复数形式就不太正确了。实际上，Spring Data REST将“taco”的复数形式计算成了“tacoes”，所以，为了向taco发送请求，我们必须将错就错，请求“/api/tacoes”地址：

```
% curl localhost:8080/api/tacoes
{
  "_embedded" : {
    "tacoes" : [ {
      "name" : "Carnivore",
      "createdAt" : "2018-02-11T17:01:32.999+0000",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "taco" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "ingredients" : {
          "href" : "http://localhost:8080/api/tacoes/2/ingredients"
        }
      }
    }]
  },
  "page" : {
    "size" : 20,
    "totalElements" : 3,
    "totalPages" : 1,
    "number" : 0
  }
}
```

你肯定会想，我是怎么知道“taco”的复数形式被错误计算成了“tacoes”。实际上，Spring Data REST还暴露了一个主资源（home resource），这个资源包含了所有端点的链接。我们只需要向API的基础路径发送GET请求就能得到它的结果：

```
$ curl localhost:8080/api
{
  "_links" : {
    "orders" : {
      "href" : "http://localhost:8080/api/orders"
    },
    "ingredients" : {
      "href" : "http://localhost:8080/api/ingredients"
    },
    "tacoes" : {
      "href" : "http://localhost:8080/api/tacoes{?page,size,sort}",
      "templated" : true
    },
    "users" : {
      "href" : "http://localhost:8080/api/users"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile"
    }
  }
}
```

可以看到，这个主资源显示了所有实体的链接。除了tacoes链接之外，其他都很好，在这里关系名和URL地址上都是“taco”错误的复数形式。

好消息是，我们并非必须接受Spring Data REST的这个小错误。通过为Taco添加一个简单的注解，我们就能调整关系名和路径：

```
@Data
@Entity
@RestResource(rel="tacos", path="tacos")
public class Taco {
  ...
}
```

@RestResource注解能够为实体提供任何我们想要的关系名和路径。在本例中，我们将它们都设置成了“tacos”。现在，我们请求主资源

的时候，会看到tacos有了正确的复数形式：

```
"tacos" : {
  "href" : "http://localhost:8080/api/tacos{?page,size,sort}",
  "templated" : true
},
```

这样就将我们的端点路径整理好了，现在可以向“/api/tacos”发送请求来操作taco资源了。

接下来我们看一下使用Spring Data REST端点如何进行排序。

6.3.2 分页和排序

你可能已经发现，主资源上的所有链接都提供了可选的page、size和sort参数。默认情况下，请求集合资源（比如“/api/tacos”）都会返回第一页的20个条目。但是，我们可以通过在请求中指定page和size参数调整具体的页数和每页的数量。

例如，我们想要请求第一页的taco，但是希望包含5个条目，就可以发送如下的GET请求（使用curl）：

```
$ curl "localhost:8080/api/tacos?size=5"
```

如果taco的数量超过了5个，我们可以通过使用page参数获取第二页的taco：

```
$ curl "localhost:8080/api/tacos?size=5&page=1"
```

注意，page参数是从0开始计算的，也就是说page值为1的时候会请求第二页的数据（你可能会发现，很多命令行shell遇到请求中的&符号会出错，所以我们在前面的curl命令中为整个URL使用了引号）。

我们可以使用字符串操作来将这些参数拼接到URL上，但是HATEOAS为我们提供了第一页、下一页、上一页和最后一页等链接：

```
"_links" : {
  "first" : {
    "href" : "http://localhost:8080/api/tacos?page=0&size=5"
  },
  "self" : {
    "href" : "http://localhost:8080/api/tacos"
  },
  "next" : {
    "href" : "http://localhost:8080/api/tacos?page=1&size=5"
  },
  "last" : {
    "href" : "http://localhost:8080/api/tacos?page=2&size=5"
  },
  "profile" : {
    "href" : "http://localhost:8080/api/profile/tacos"
  },
  "recents" : {
    "href" : "http://localhost:8080/api/tacos/recent"
  }
}
```

有了这些链接，API的客户端就不需要跟踪当前正处于哪一页，也不用将参数自己拼接到URL上了，只需根据名字查找这些页面导航并进行访问就可以了。

sort参数允许我们根据实体的某个属性对结果进行排序。例如，我们想要获取最近创建的12条taco进行UI展示，就可以混合使用分页和排序参数实现：

```
$ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"
```

在这里，sort参数指定我们要按照createdAt属性进行排序，并且要按照降序进行排列（所以最新的taco会放在最前面）。page和size参数指定我们想要获取第一页的12个taco。

这恰好是UI展现最近创建的taco所需要的数据。它与我们在本章前文DesignTaco Controller定义的“/design/recent”端点大致相同。

这里还有一个小问题。UI代码需要硬编码才能请求带有指定参数的taco列表。当然，这可以正常运行。但是，如果让客户端太多地了解如何构建API请求，就会在一定程度上增加脆弱性。如果客户端能够从链接列表中查找URL就太好了。如果URL能够更简洁，就像前面看到“/design/recent”一样，那就更棒了。

6.3.3 添加自定义的端点

Spring Data REST能够很好地为执行CRUD操作的Spring Data repository创建端点。但有时候，我们需要脱离默认的CRUD API，创建处理核心问题的端点。

我们当然可以在带有@RestController注解的bean中实现任意的端点，以此来补充Spring Data REST端点的不足。实际上，我们可以重新启用本章前面提到的DesignTacoController，它依然可以与Spring Data REST提供的端点一起运行。

但是在编写自己的API控制器时，它们的端点在有些方面会与Spring Data REST的端点脱节：

- 我们自己的控制器端点没有映射到Spring Data REST的基础路径下。虽然我们可以强制要求它们映射到任意前缀作为基础路径，包括使用Spring Data REST的基础路径，但是如果基础路径发生变化的话，我们还需要修改控制器的映射，以便于与其匹配。
- 在自己控制器所定义的端点中，返回资源时并不会自动包含超链接，与Spring Data REST所返回的资源是不同的。这意味着，客户端无法通过关系名发现自定义的端点。

我们首先来解决基础路径的问题。Spring Data REST提供了一个新的注解@RepositoryRestController，这个注解可以用到控制器类上，这样控制器类所有映射的基础路径就会与Spring Data REST端点配置的基础路径相同。简而言之，在使用@RepositoryRestController注解的控制器中，所有映射将会具有和spring.data.rest.base-path属性值一样的前缀（我们之前将这个属性的值配置成了“/api”）。

在这里我们会创建一个只包含recentTacos()方法的新控制器，而不是修改已有的DesignTacoController，因为这个旧的控制器包含了多个我们不再需要的方法。程序清单6.7中的RecentTacosController带有@RepositoryRestController注解，表明它会将Spring Data REST的基础路径用到自己的请求映射上。

程序清单6.7　将Spring Data REST的基础路径用到控制器上

```
package tacos.web.api;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
```

```java
import java.util.List;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import tacos.Taco;
import tacos.data.TacoRepository;

@RepositoryRestController
public class RecentTacosController {

  private TacoRepository tacoRepo;

  public RecentTacosController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping(path="/tacos/recent", produces="application/hal+json")
  public ResponseEntity<Resources<TacoResource>> recentTacos() {
    PageRequest page = PageRequest.of(
                          0, 12, Sort.by("createdAt").descending());
    List<Taco> tacos = tacoRepo.findAll(page).getContent();

    List<TacoResource> tacoResources =
        new TacoResourceAssembler().toResources(tacos);
    Resources<TacoResource> recentResources =
            new Resources<TacoResource>(tacoResources);
    recentResources.add(
        linkTo(methodOn(RecentTacosController.class).recentTacos())
            .withRel("recents"));
    return new ResponseEntity<>(recentResources, HttpStatus.OK);
  }

}
```

虽然@GetMapping映射到了“/tacos/recent”路径，但是类级别的@Repository RestController注解会确保这个路径添加Spring Data REST的基础路径作为前缀。按照我们的配置，recentTacos()方法将会处理针对“/api/tacos/recent”的GET请求。

需要注意的是，尽管@RepositoryRestController的名称和@RestController非常相似，但是它并没有和@RestController相同的语义。具体来讲，它并不能保证处理器方法返回的值会自动写入响应体中。所以，我们要么为方法添加@ResponseBody注解，要么返回包装响应数据的ResponseEntity。这里我们选择的方案是返回ResponseEntity。

RecentTacosController准备就绪之后，对“/api/tacos/recent”发送请求最多会返回15条最近创建的taco，此时就不需要在URL中添加分页和排序参数了。但是在请求“/api/tacos”的时候，这个地址依然没有出现在结果的超链接列表中。

6.3.4　为Spring Data端点添加自定义的超链接

如果最近taco端点没有出现在“/api/tacos”所返回的超链接中，那么客户端如何知道该怎样获取最近的taco呢？它要么猜测，要么使用分页和排序参数。无论采用哪种方式，都需要在客户端代码中硬编码，这都不是理想的做法。

通过声明资源处理器（resource processor）bean，我们可以为Spring Data REST自动包含的链接列表继续添加链接。Spring Data HATEOAS提供了一个ResourceProcessor接口，能够在资源通过API返回之前对其进行操作。对于我们的场景来说，需要一个 ResourceProcessor 实现，为PagedResources<Resource<Taco>>类型的资源（也就是“/api/tacos”端点所返回的类型）添加recents链接。程序清单6.8展现了定义这种

ResourceProcessor的bean声明方法。

程序清单6.8　为Spring Data REST端点添加自定义的链接

```
@Bean
public ResourceProcessor<PagedResources<Resource<Taco>>>
  tacoProcessor(EntityLinks links) {

  return new ResourceProcessor<PagedResources<Resource<Taco>>>() {
    @Override
    public PagedResources<Resource<Taco>> process(
                        PagedResources<Resource<Taco>> resource) {
      resource.add(
          links.linkFor(Taco.class)
               .slash("recent")
               .withRel("recents"));
      return resource;
    }
  };
}
```

程序清单6.8中的ResourceProcessor定义了一个匿名内部类，并将其声明为Spring应用上下文中所创建的bean。Spring HATEOAS会自动发现这个bean（以及其他ResourceProcessor类型的bean）并将其应用到对应的资源上。在本例中，如果控制器返回PagedResources<Resource<Taco>>，就会包含一个最近创建的taco链接。这就包括了对“/api/tacos”请求的响应。

6.4　小结

- REST端点可以通过Spring MVC来创建，这里的控制器与面向浏览器的控制器遵循相同的编程模型。
- 为了绕过视图和模型的逻辑并将数据直接写入响应体中，控制器处理方法既可以添加@ResponseBody注解也可以返回ResponseEntity对

象。

- @RestController注解简化了REST控制器，使用它的话，处理器方法中就不需要添加@ResponseBody注解了。
- Spring HATEOAS为Spring MVC控制器返回的资源启用了超链接功能。
- 借助Spring Data REST，Spring Data repository可以自动导出为REST API。

---

[1]　在这里，我选择使用Angular，但是前端框架的选择应该对后端Spring代码的编写没有影响，所以你尽可以选择Angular、React、Vue.js或者其他适合你的前端技术。

第7章　消费REST服务

本章内容：

- 使用RestTemplate消费REST API
- 使用Traverson导航超媒体API

你有没有过这样的经历——跑去看电影，却发现自己是影院中唯一的一个人。这当然是一种很奇妙的经历，从本质上来讲，这变成了一个私人电影。你可以选择任意想要的座位、和屏幕上的角色交谈，甚至可以打开手机发推特，完全不用担心因为破坏了别人的观影体验而惹人生气。最棒的是，没有人会毁了你观看这部电影的心情。

对我来说，这样的事情并不常见。但是，遇到这种情况的时候，我会想，如果我也不出现的话会发生什么呢？工作人员还会播放这部影片吗？电影中的英雄还会拯救世界吗？电影播放结束后，工作人员还会打扫影院吗？

没有观众的电影就像没有客户端的API。这些API已经准备好接收

和提供数据了，但是如果它们从来没有被调用过，它们还是API吗？就像薛定谔的猫一样，在发起请求之前，我们并不知道这个API是否活跃，也不知道它是否返回HTTP 404响应。

在前面的章节中，我们主要关注如何定义REST端点，它们可以被应用外部的客户端所消费。尽管开发这种API的主要驱动力是单页Angular应用，以便于实现Taco Cloud Web站点，但实际上客户端可以是任意应用，可以是任何语言，甚至可以是另外一个Java应用。

Spring应用除了提供对外API之外，同时要对另外一个应用的API发起请求的场景并不罕见。实际上，在微服务领域，这正变得越来越普遍。因此，花点时间研究一下如何使用Spring与REST API交互是非常值得的。

Spring应用可以采用多种方式来消费REST API，包括以下几种方式：

- RestTemplate：Spring核心框架提供的简单、同步REST客户端。
- Traverson：Spring HATEOAS提供的支持超链接、同步的REST客户端，其灵感来源于同名的JavaScript库。
- WebClient：Spring 5所引入的反应式、异步REST客户端。

我将WebClient推迟到第11章讨论Spring的反应式Web框架时再进行介绍，现在我们主要关注其他的两个REST客户端。下面先从RestTemplate开始。

7.1 使用RestTemplate消费REST端点

从客户端的角度来看，与REST资源进行交互涉及很多工作，而且大多数都是很单调乏味的样板式代码。如果使用较低层级HTTP库，客户端就需要创建一个客户端实例和请求对象、执行请求、解析响应、将响应映射为领域对象，并且还要处理这个过程中可能会抛出的所有异常。不管发送什么样的HTTP请求，这种样板代码都要不断重复。

为了避免这种样板代码，Spring提供了RestTemplate。就像JDBCTemplate能够处理JDBC中丑陋的那部分代码一样，RestTemplate也能够将你从消费REST资源所面临的单调工作中解放出来。

RestTemplat提供了41个与REST资源交互的方法。我们不会详细介绍它所提供的所有方法，而是只考虑12个独立的操作（见表7.1），每种方法都有重载形式，它们组成了完整的41个方法。

表**7.1　RestTemplate**中**12**个独立的操作

方法	描述
delete(…)	在特定的URL上对资源执行HTTP DELETE操作
exchange(…)	在URL上执行特定的HTTP方法，返回包含对象的ResponseEntity，这个对象是从响应体中映射得到的
execute(…)	在URL上执行特定的HTTP方法，返回一个从响应体映射得到的对象

getForEntity(…)	发送一个HTTP GET请求，返回的ResponseEntity包含了响应体所映射成的对象
getForObject(…)	发送一个HTTP GET请求，返回的请求体将映射为一个对象
headForHeaders(…)	发送HTTP HEAD请求，返回包含特定资源URL的HTTP头信息
optionsForAllow(…)	发送HTTP OPTIONS请求，返回特定URL的Allow头信息
patchForObject(...)	发送HTTP PATCH请求，返回一个从响应体映射得到的对象
postForEntity(…)	POST数据到一个URL，返回包含一个对象的ResponseEntity，这个对象是从响应体中映射得到的
postForLocation(…)	POST数据到一个URL，返回新创建资源的URL
postForObject(…)	POST数据到一个URL，返回根据响应体匹配形成的对象
put(…)	PUT资源到特定的URL

除了TRACE以外，RestTemplate对每种标准的HTTP方法都提供了至少一个方法。除此之外，execute()和exchange()提供了较低层次的通用方法，以便于进行任意的HTTP操作。

表7.1中的大多数操作都以3种方法的形式进行了重载。

- 使用String作为URL格式，并使用可变参数列表指明URL参数。
- 使用String作为URL格式，并使用Map<String,String>指明URL参数。
- 使用java.net.URI作为URL格式，不支持参数化URL。

明确了RestTemplate所提供的12个操作以及各个变种如何工作之后，你就能以自己的方式编写消费REST资源的客户端了。

要使用RestTemplate，你可以在需要的地方创建一个实例：

```
RestTemplate rest = new RestTemplate();
```

也可以将其声明为一个bean并注入到需要的地方：

```
@Bean
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

我们通过对4个主要HTTP方法（GET、PUT、DELETE和POST）的支持来研究RestTemplate的操作。下面我们从GET方法的getForObject()和getForEntity()开始。

7.1.1　GET资源

假设我们现在想要通过Taco Cloud API获取某个配料，并且API没有实现HATEOAS，那么我们可以使用getForObject()获取配料。例如，如下的代码将使用RestTemplate根据ID来获取Ingredient对象：

```
public Ingredient getIngredientById(String ingredientId) {
```

```
  return rest.getForObject("http://localhost:8080/ingredients/{id}",
                           Ingredient.class, ingredientId);
}
```

在这里，我们使用了getForObject()的变种形式，接收一个String类型的URL并使用可变列表来指定URL变量。传递给getForObject()的ingredientId参数会用来填充给定URL的{id}占位符。尽管在本例中只有一个URL变量，但是很重要的一点需要我们注意，变量参数会按照它们出现的顺序设置到占位符中。getForObject()方法的第二个参数是响应应该绑定的类型。在本例中，响应数据（很可能是JSON格式）应该被反序列化为要返回的Ingredient对象。

另外一种替代方案是使用Map来指定URL变量：

```
public Ingredient getIngredientById(String ingredientId) {
  Map<String,String> urlVariables = new HashMap<>();
  urlVariables.put("id", ingredientId);
  return rest.getForObject("http://localhost:8080/ingredients/{id}",
                           Ingredient.class, urlVariables);
}
```

在本例中，ingredientId的值会被映射到名为id的key上。当发起请求的时候，{id}占位符将会被替换为key为id的Map条目上。

使用URL参数要稍微复杂一些，这种方式需要我们在调用getForObject()之前构建一个URI对象。在其他方面，它与另外两个变种非常类似：

```
public Ingredient getIngredientById(String ingredientId) {
  Map<String,String> urlVariables = new HashMap<>();
  urlVariables.put("id", ingredientId);
  URI url = UriComponentsBuilder
            .fromHttpUrl("http://localhost:8080/ingredients/{id}")
```

```
            .build(urlVariables);

  return rest.getForObject(url, Ingredient.class);
}
```

在这里，URI对象是通过String规范定义的，它的占位符会被Map中的条目所替换，与之前看到的getForObject()变种非常相似。getForObject()是获取资源的有效方式，但是如果客户端需要的不仅仅是载荷体，那么可以考虑使用getForEntity()。

getForEntity()的工作方式和getForObject()类似，但是它所返回的并不是代表响应载荷的领域对象，而是一个包裹领域对象的ResponseEntity对象。借助ResponseEntity对象能够访问很多的响应细节，比如响应头信息。

例如，我们除了想要获取配料数据之外，还想要从响应中探查Date头信息。借助getForEntity()，这个需求很容易实现：

```
public Ingredient getIngredientById(String ingredientId) {
  ResponseEntity<Ingredient> responseEntity =
      rest.getForEntity("http://localhost:8080/ingredients/{id}",
          Ingredient.class, ingredientId);

  log.info("Fetched time: " +
          responseEntity.getHeaders().getDate());

  return responseEntity.getBody();
}
```

getForEntity()有与getForObject()方法相同参数的重载形式，所以我们既可以以可变列表参数的形式提供URL变量，也可以以URI对象的形式调用getForEntity()。

7.1.2　PUT资源

为了发送HTTP PUT请求，RestTemplate提供了put()方法。put()方法的3个变种形式都会接收一个Object，它会被序列化并发送至给定的URL。就URL本身来讲，它可以以URI对象或String的形式来指定。与getForObject()和getForEntity()类似，URL变量能够以可变参数列表或Map的形式来提供。

假设我们想要使用一个新Ingredient对象的数据来替换某个配料资源，那么如下的代码片段就能做到这一点：

```
public void updateIngredient(Ingredient ingredient) {
  rest.put("http://localhost:8080/ingredients/{id}",
           ingredient,
           ingredient.getId());
}
```

在这里，URL是以String的形式指定的。该URL包含一个占位符，它会被给定Ingredient的id属性所替换。要发送的数据是Ingredient对象本身。put()方法返回的是void，所以没有必要处理返回值。

7.1.3　DELETE资源

假设Taco Cloud不再想要提供某种配料，因此要从可选列表中将其完全删除。为了实现这一点，我们可以使用RestTemplate来调用delete()方法：

```
public void deleteIngredient(Ingredient ingredient) {
```

```
  rest.delete("http://localhost:8080/ingredients/{id}",
              ingredient.getId());
}
```

在本例中，我们只为delete()提供了URL（以String的形式指定）和URL变量值。但是，和其他的RestTemplate方法类似，URL能够以URI对象的方式来指定，并且URL参数也能够以Map的方式来声明。

7.1.4　POST资源

现在，假设要添加新的配料到Taco Cloud菜单中，我们可以向“.../ingredients”端点发送HTTP POST请求并将配料数据放到请求体中。RestTemplate有3种发送POST请求的方法，每种方法都有相同的重载变种来指定URL。如果你希望在POST请求之后得到新创建的Ingredient资源，那么可以按照如下方式使用postForObject()：

```
public Ingredient createIngredient(Ingredient ingredient) {
  return rest.postForObject("http://localhost:8080/ingredients",
                            ingredient,
                            Ingredient.class);
}
```

postForObject()方法的这个变种形式接收一个String类型的URL规范、要提交给服务器端的对象以及响应体应该绑定的领域类型。尽管我们在这里没有用到，但是第4个参数可以是URL变量值的Map或者是可变参数的列表，它们能够替换到URL之中。

如果客户端还想要知道新创建资源的地址，那么我们可以调用postForLocation()方法：

```
public URI createIngredient(Ingredient ingredient) {
  return rest.postForLocation("http://localhost:8080/ingredients",
                              ingredient);
}
```

注意，postForLocation()的工作方式与postForObject()类似，只不过它所返回的是新创建资源的URI，而不是资源对象本身。这里所返回的URI是从响应的Location头信息中派生出来的。如果你同时需要地址和响应载荷，那么可以使用postForEntity()方法：

```
public Ingredient createIngredient(Ingredient ingredient) {
  ResponseEntity<Ingredient> responseEntity =
         rest.postForEntity("http://localhost:8080/ingredients",
                            ingredient,
                            Ingredient.class);

  log.info("New resource created at " +
           responseEntity.getHeaders().getLocation());

  return responseEntity.getBody();
}
```

尽管RestTemplate的方法在目的上有所不同，但是它们的用法非常相似。因此，我们很容易就可以精通RestTemplate并将其用到客户端代码中。

另一方面，如果你所消费的API在响应中包含了超链接，那么RestTemplate就力所不及了。当然，我们可以使用RestTemplate获取更详细的资源数据，然后处理里面所包含的内容和链接，但是这个任务并不简单。与其使用RestTemplate来处理超媒体API，还不如选择一个专门关注该领域的库，那就是Traverson。

7.2 使用Traverson导航REST API

Traverson来源于Spring Data HATEOAS项目，是Spring应用中开箱即用的消费超媒体API的解决方案。这个基于Java的库灵感来源于同名的JavaScript库。

你可能已经发现Traverson的名字有点类似于“traverse on”，这种叫法其实可以很好地描述它的用法。在本节中，我们将会以遍历API关系名的方式来消费API。

要使用Traverson，首先我们要用API的基础URI来实例化一个Traverson对象：

```
Traverson traverson = new Traverson(
    URI.create("http://localhost:8080/api"), MediaTypes.HAL_JSON);
```

在这里，我将Traverson指向了Taco Cloud的基础URL（本地运行）。这是需要给Traverson指定的唯一URL。从这里开始，我们就可以根据链接的关系名来遍历API。我们同时还指定了API将会生成JSON格式的响应，并且具有HAL风格的超链接，这样Traverson就能知道怎样解析传入的资源数据了。与RestTemplate类似，你可以选择在使用Traverson对象之前实例化它，也可以将其声明为一个bean并在需要的地方注入进来。

有了Traverson对象之后，我们就可以通过以下链接使用API。例如，假设我们想检索所有配料的列表。从6.3.1小节可以知道，配料链接

有一个href属性，它会链接到配料资源。我们需要跟踪这个链接：

```
ParameterizedTypeReference<Resources<Ingredient>> ingredientType =
    new ParameterizedTypeReference<Resources<Ingredient>>() {};

Resources<Ingredient> ingredientRes =
    traverson
      .follow("ingredients")
      .toObject(ingredientType);

Collection<Ingredient> ingredients = ingredientRes.getContent();
```

通过调用Traverson对象的follow()方法，我们就可以导航至链接关系名为ingredients的资源。现在，客户端已经导航至ingredients，我们需要通过调用toObject()来提取资源的内容。

我们需要告诉toObject()方法要将数据读入到哪种对象之中。考虑到我们需要以Resources<Ingredient>对象的形式来读入，而且Java类型擦除使得为泛型提供类型信息变得非常困难，所以这可能会有些棘手。但是，创建ParameterizedTypeReference能够帮助我们解决这个问题。

打个比方，假设这不是REST API，而是Web站点上的主页；这也不是REST客户端代码，而是我们正在浏览器中查看主页。在页面中，我们看到了一个关于Ingredients的链接，点击进入该链接。在进入下一个页面的时候，我们需要读取该页面，类似于Traverson将内容提取为Resources<Ingredient>对象。

现在，我们考虑一个更有趣的场景——假设我们想要获取最新创建的taco。从主资源开始，我们可以按照如下方式导航至最近的taco资源：

```
ParameterizedTypeReference<Resources<Taco>> tacoType =
    new ParameterizedTypeReference<Resources<Taco>>() {};

Resources<Taco> tacoRes =
    traverson
      .follow("tacos")
      .follow("recents")
      .toObject(tacoType);

Collection<Taco> tacos = tacoRes.getContent();
```

在这里，我们跟踪tacos链接，然后从这里开始，跟踪recents链接。通过这种方式，我们得到了感兴趣的资源，所以基于对应的ParameterizedTypeReference调用toObject()方法，我们就得到了想要的内容。我们可以通过列出关系名称列表的形式来简化“.follow()”方法：

```
Resources<Taco> tacoRes =
    traverson
      .follow("tacos", "recents")
      .toObject(tacoType);
```

正如我们所看到的，Traverson能够很容易地导航HATEOAS的API并消费其资源。但是，它并没有提供通过这些API写入或删除资源的方法。相反，RestTemplate能够写入和删除资源，但是在导航API方面支持得并不太好。

当你既要导航API又要更新或删除资源时，你需要组合使用RestTemplate和Traverson。Traverson仍然可以导航至创建新资源的链接。然后，可以将这个链接传递给RestTemplate来执行POST、PUT、DELETE或任何其他需要的HTTP请求。

例如，我们想要为Taco Cloud菜单添加一个新的Ingredient，如下的

addIngredient ()方法将Traverson和RestTemplate组合起来，向API提交了一个新的Ingredient：

```
private Ingredient addIngredient(Ingredient ingredient) {
  String ingredientsUrl = traverson
      .follow("ingredients")
      .asLink()
      .getHref();

  return rest.postForObject(ingredientsUrl,
                            ingredient,
                            Ingredient.class);
}
```

在跟踪完Ingredients之后，我们通过调用asLink()方法得到链接本身。基于该链接，我们调用getHref()得到链接的URL。有了URL之后，我们就具备了使用RestTemplate调用postForObject()并创建新配料所需的一切。

7.3 小结

- 客户端可以使用RestTemplate针对REST API发送HTTP请求。
- Traverson能够让客户端导航响应中内嵌超链接的API。

第8章　发送异步消息

本章内容：

- 异步化的消息
- 使用JMS、RabbitMQ和Kafka发送消息
- 从代理拉取消息
- 监听消息

在星期五下午4点55分，再有几分钟你就可以开始休假了。现在，你的时间只够开车到机场赶上航班。但是在你离开办公室之前，你需要确定老板和同事了解你目前的工作进展，这样他们就可以在星期一继续完成你留下的工作。不过，你的一些同事已经提前离开过周末去了，而你的老板正在忙于开会。你该怎么办呢？

要想既传达到你的工作状态又能赶上飞机，最有效的方式就是发送一封电子邮件给你的老板和同事，详述工作进展并且承诺给他们寄张明信片。你不知道他们在哪里，也不知道他们什么时候才能真正读到你的

邮件。但是你知道，他们终究会回到他们的办公桌旁，阅读你的邮件。而此时，你可能正在赶往机场的路上。

同步通信，比如我们在前面所看到的REST，有它自己的适用场景。不过，对于开发者而言，这种通信方式并不是应用程序之间进行交互的唯一方式。异步消息是一个应用程序向另一个应用程序间接发送消息的一种方式，这种间接性能够为进行通信的应用带来更松散的耦合和更大的可伸缩性。

在本章中，我们将会使用异步消息从Taco Cloud Web站点发送订单信息到一个单独的应用中，这个应用是Taco Cloud的厨房，在这里会烹制taco。我们将会考虑Spring提供的3种异步消息方案：Java消息服务（Java Message Service，JMS）、RabbitMQ和高级消息队列协议（Advanced Message Queueing Protocol）、Apache Kafka。除了基础的发送和接收消息之外，我们还会看一下Spring对消息驱动POJO的支持，它是一种与EJB的消息驱动Bean（Message-Driven Bean，MDB）类似的消息接收方式。

8.1 使用JMS发送消息

JMS是一个Java标准，定义了使用消息代理（message broker）的通用API，最早于2001年提出。长期以来，JMS一直是实现异步消息的首选方案。在JMS出现之前，每个消息代理都有私有的API，这就使得不同代理之间的消息代码很难通用。但是借助JMS，所有遵从规范的实现都使用通用的接口，这就好像JDBC为数据库操作提供了通用的接口一

样。

Spring通过基于模板的抽象为JMS功能提供了支持，这个模板就是JmsTemplate。借助JmsTemplate，我们能够非常容易地在消息生产方发送队列和主题消息，在消费消息的那一方，也能够非常容易地接收这些消息。Spring还提供了消息驱动POJO的理念：这是一个简单的Java对象，它能够以异步的方式响应队列或主题上到达的消息。

我们将会讨论Spring对JMS的支持，包括JmsTemplate和消息驱动POJO。但是在发送和接收消息之前，我们首先需要一个消息代理（broker），它能够在消息的生产者和消费者之间传递消息。对Spring JMS的探索就从在Spring中搭建消息代理开始吧。

8.1.1　搭建JMS环境

在使用JMS之前，我们必须要将JMS客户端添加到项目的构建文件中。借助Spring Boot，这再简单不过了。我们所需要做的就是添加一个starter依赖到构建文件中。但是，首先，我们需要决定该使用Apache ActiveMQ还是更新的Apache ActiveMQ Artemis代理。

如果选择使用Apache ActiveMQ，那么我们需要添加如下的依赖到项目的pom.xml文件中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

如果选择使用ActiveMQ Artemis，那么starter依赖将会如下所示：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
```

Artemis是重新实现的下一代ActiveMQ，使ActiveMQ变成了遗留方案。因此，Taco Cloud会选择使用Artemis。但是在编写发送和接收消息的代码方面，选择哪种方案几乎没有什么影响。唯一需要注意的重要差异就是如何配置Spring创建到代理的连接。

默认情况下，Spring会假定Artemis代理在localhost的61616端口运行。对于开发来说，这样是没有问题的，但是一旦要将应用部署到生产环境，我们就需要设置一些属性来告诉Spring该如何访问代理。最有用的属性如表8.1所示。

表**8.1**　配置**Artemis**代理的位置和凭证信息的属性

属性	描述
spring.artemis.host	代理的主机
spring.artemis.port	代理的端口
spring.artemis.user	用来访问代理的用户（可选）
spring.artemis.password	用来访问代理的密码（可选）

例如，我们看看如下的application.yml文件条目，它可能会用于一个非开发的环境：

```
spring:
  artemis:
    host: artemis.tacocloud.com
    port: 61617
    user: tacoweb
    password: l3tm31n
```

这会让Spring创建到Artemis代理的连接，该Artemis代理监听artemis.tacocloud.com的61617端口。它还为应用设置了与代理交互的凭证信息。凭证信息是可选的，但是对于生产环境来说，我们推荐使用它们。

如果你选择使用ActiveMQ而不是Artemis，那么需要使用ActiveMQ特定的属性，如表8.2所示。

表**8.2**　配置**ActiveMQ**代理的位置和凭证信息的属性

属性	描述
spring.activemq.broker-url	代理的URL
spring.activemq.user	用来访问代理的用户（可选）
spring.activemq.password	用来访问代理的密码（可选）
spring.activemq.in-memory	是否启用在内存中运行的代理（默认为true）

需要注意，ActiveMQ代理不是分别设置代理的主机和端口，而是使用了一个名为spring.activemq.broker-url的属性来指定代理的地址。URL应该是“tcp://”协议的地址，如下面的YMAL片段所示：

```
spring:
  activemq:
    broker-url: tcp://activemq.tacocloud.com
    user: tacoweb
    password: l3tm31n
```

不管你是选择Artemis还是ActiveMQ，如果是在本地开发运行，那么你都不需要配置这些属性。

但是，如果你选择使用ActiveMQ，需要将spring.activemq.in-memory属性设置为false，防止Spring启动内存中运行的代理。内存中运行的代理看起来很有用，但是只有同一个应用发布和消费消息时才能使用它（这限制了它的用途）。

在继续下面的内容之前，我们要安装并启动一个Artemis（或ActiveMQ）代理，而不是选择使用嵌入式的代理。我在这里不再重复讲述安装过程，推荐你参考Artemis和ActiveMQ的文档来了解详细内容。

现在，我们已经在构建文件中添加了对JMS starter的依赖，代理也已经准备好将消息从一个应用传递到另一个应用。接下来，我们就可以开始发送消息了。

8.1.2　使用JmsTemplate发送消息

将JMS starter依赖（不管是Artemis还是ActiveMQ）添加到构建文件之后，Spring Boot会自动配置一个JmsTemplate（以及其他内容），我们可以将它注入到其他bean中，并使用它来发送和接收消息。

JmsTemplate是Spring对JMS集成支持功能的核心。与Spring其他面向模板的组件类似，JmsTemplate消除了大量传统使用JMS时所需的样板代码。如果没有JmsTemplate的话，那么我们需要编写代码来创建到消息代理的连接和会话，还要编写更多的代码来处理发送消息过程中可能出现的异常。JmsTemplate能够让我们关注真正要做的事情：发送消息。

JmsTemplate有多个用来发送消息的方法，包括：

```
// 发送原始的消息
void send(MessageCreator messageCreator) throws JmsException;
void send(Destination destination, MessageCreator messageCreator)
                                                throws JmsException;
void send(String destinationName, MessageCreator messageCreator)
                                                throws JmsException;
// 发送根据对象转换而成的消息
void convertAndSend(Object message) throws JmsException;
void convertAndSend(Destination destination, Object message)
                                                throws JmsException;
void convertAndSend(String destinationName, Object message)
                                                throws JmsException;

// 发送根据对象转换而成的消息并且带有后期处理的功能
void convertAndSend(Object message,
            MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(Destination destination, Object message,
            MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(String destinationName, Object message,
            MessagePostProcessor postProcessor) throws JmsException;
```

我们可以看到，实际上只有两个方法，也就是send()和

convertAndSend()，每个方法都有重载形式以支持不同的参数。如果我们仔细观察一下，就会发现convertAndSend()的各种形式又可以分成两个子类型。在考虑这些方法作用的时候，我们对它们进行以下细分。

- 3个send()方法都需要MessageCreator来生成Message对象。
- 3个convertAndSend()方法会接受Object对象，并且会在幕后自动将Object转换为Message。
- 3个convertAndSend()会自动将Object转换为Message，但同时还能接受一个MessagePostProcessor对象，用来在发送之前对Message进行自定义。

这 3 种方法分类都分别包含 3 个重载方法，它们的区别在于如何指定JMS的目的地（队列或主题）。

- 有1个方法不接受目的地参数，它会将消息发送至默认的目的地。
- 有1个方法接受Destination对象，该对象指定了消息的目的地。
- 有1个方法接受String，它通过名字的形式指定了消息的目的地。

要让这些方法真正发挥作用，我们看一下程序清单 8.1 中的JmsOrderMessaging Service，它使用了形式最简单的send()方法。

程序清单8.1　使用.send()方法将订单发送至默认的目的地

```
package tacos.messaging;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Service;
```

```
@Service
public class JmsOrderMessagingService implements OrderMessagingService {
  private JmsTemplate jms;

  @Autowired
  public JmsOrderMessagingService(JmsTemplate jms) {
    this.jms = jms;
  }

  @Override
  public void sendOrder(Order order) {
    jms.send(new MessageCreator() {
                @Override
                public Message createMessage(Session session)
                                              throws JMSException {
                  return session.createObjectMessage(order);
                }
              }
    );
  }
}
```

sendOrder()方法调用了jms.send()，并传递了MessageCreator接口的一个匿名内部实现。这个实现类重写了createMessage()方法，从而能够通过给定的Order对象创建新的对象消息。

我不知道你的感觉如何，但是我认为程序清单8.1虽然比较直接，但还是有点啰唆。声明匿名内部类的过程使得原本很简单的方法调用变得很复杂。我们发现MessageCreator是一个函数式接口，所以我们可以通过lambda表达式简化一下sendOrder()方法：

```
@Override
public void sendOrder(Order order) {
  jms.send(session -> session.createObjectMessage(order));
}
```

但是，需要注意对jms.send()的调用并没有指定目的地。为了让它能

够运行，我们需要通过名为spring.jms.template.default-destination的属性声明一个默认的目的地名称。例如，我们可以在application.yml文件中这样设置该属性：

```
spring:
  jms:
    template:
      default-destination: tacocloud.order.queue
```

在很多场景下，使用默认的目的地是最简单的可选方案。借助它，我们只声明一次目的地名称就可以了，代码只关心发送消息，而无须关心消息会发到哪里。但是，如果我们要将消息发送至默认目的地之外的其他地方，那么我们需要通过为send()设置参数来进行指定。

其中一种方式就是传递Destination对象作为send()方法的第一个参数。最简单的方式就是声明一个Destination bean并将其注入处理消息的bean中。例如，如下的bean声明Taco Cloud订单队列的Destination：

```
@Bean
public Destination orderQueue() {
  return new ActiveMQQueue("tacocloud.order.queue");
}
```

很重要的一点需要注意，这里的ActiveMQQueue来源于Artemis（来自org.apache. activemq.artemis.jms.client包）。如果选择使用ActiveMQ（而不是Artemis），那么同样有一个名为ActiveMQQueue的类（来自org.apache.activemq.command包）。

在Destination bean注入到JmsOrderMessagingService之后，调用send()的时候，我们就可以使用它来指定目的地了：

```
private Destination orderQueue;

@Autowired
public JmsOrderMessagingService(JmsTemplate jms,
                                Destination orderQueue) {
  this.jms = jms;
  this.orderQueue = orderQueue;
}

...

@Override
public void sendOrder(Order order) {
  jms.send(
      orderQueue,
      session -> session.createObjectMessage(order));
}
```

通过Destination指定目的地的时候，我们其实可以设置Destination的更多属性，而不仅仅是目的地的名称。但是，在实践中，除了目的地名称我们几乎不会设置其他的属性。因此，使用名称作为send()的第一个参数会更加简单：

```
@Override
public void sendOrder(Order order) {
  jms.send(
      "tacocloud.order.queue",
      session -> session.createObjectMessage(order));
}
```

尽管send()方法使用起来并不是特别困难（尤其是通过lambda表达式来实现MessageCreator的时候更是如此），但是它要求我们提供MessageCreator还是增加了一点复杂性。如果我们能够只指定要发送的对象（以及可能要用到的目的地），那岂不是更简单？这其实就是convertAndSend()的工作原理。接下来，我们看一下这种方式。

消息发送之前进行转换

JmsTemplates的convertAndSend()方法简化了消息的发布，因为它不再需要MessageCreator。我们将要发送的对象直接传递给convertAndSend()，这个对象在发送之前会被转换成Message。

例如，在如下重新实现的sendOrder()方法中，使用convertAndSend()将Order对象发送到给定名称的目的地：

```
@Override
public void sendOrder(Order order) {
  jms.convertAndSend("tacocloud.order.queue", order);
}
```

与send()方法类似，convertAndSend()将会接受一个Destination对象或String值来确定目的地，我们也可以完全忽略目的地，将消息发送到默认目的地上。

不管使用哪种形式的convertAndSend()，传递给convertAndSend()的Order都会在发送之前转换成Message。在底层，这是通过MessageConverter的实现类来完成的，它替我们做了将对象转换成Message的脏活累活。

配置消息转换器

MessageConverter是Spring定义的接口，只有两个需要实现的方法：

```
public interface MessageConverter {
  Message toMessage(Object object, Session session)
                  throws JMSException, MessageConversionException;
```

```
  Object fromMessage(Message message)
}
```

尽管这个接口实现起来很简单，但我们通常并没有必要创建自定义的实现。Spring已经提供了多个实现，如表8.3所示。

表**8.3　Spring**为通用的转换任务提供了多个消息转换器（所有的消息转换器都位于**org.springframework.jms.support.converter**包中）

消息转换器	功能
MappingJackson2MessageConverter	使用Jackson 2 JSON库实现消息与JSON格式之间的相互转换
MarshallingMessageConverter	使用JAXB库实现消息与XML格式之间的相互转换
MessagingMessageConverter	使用底层的MessageConverter实现消息抽象的Message载荷与javax.jms.Message之间的转换，同时会使用JmsHeaderMapper实现JMS头信息与标准消息头信息之间的转换
SimpleMessageConverter	实现String与TextMessage之间的相互转换、字节数组与BytesMessage之间的相互转换、Map与MapMessage之间的相互转换以及Serializable对象与ObjectMessage之间的相互转换

默认情况下，将会使用SimpleMessageConverter，但是它需要被发

送的对象实现Serializable。这种办法可能也不错，但有时候我们可能想要使用其他的消息转换器来消除这种限制，比如MappingJackson2MessageConverter。

为了使用不同的消息转换器，我们必须要做的事情就是将选中的消息转换器实例声明为一个bean。例如，如下的bean声明将会使用MappingJackson2MessageConverter替代SimpleMessageConverter：

```
@Bean
public MappingJackson2MessageConverter messageConverter() {
  MappingJackson2MessageConverter messageConverter =
                        new MappingJackson2MessageConverter();
  messageConverter.setTypeIdPropertyName("_typeId");
  return messageConverter;
}
```

需要注意，在返回之前，我们调用了MappingJackson2MessageConverter的setTypeId PropertyName()方法。这非常重要，因为这样能够让接收者知道传入的消息要转换成什么类型。默认情况下，它将会包含要转换的类型的全限定类名。但是，这样的话会不太灵活，要求接收端也包含相同的类型，并且具有相同的全限定类名。

为了实现更大的灵活性，我们可以通过调用消息转换器的setTypeIdMappings()方法将一个合成类型名映射到实际类型上。举例来说，消息转换器bean方法的如下代码变更会将一个合成的order类型ID映射为Order类：

```
@Bean
public MappingJackson2MessageConverter messageConverter() {
```

```
  MappingJackson2MessageConverter messageConverter =
                         new MappingJackson2MessageConverter();
  messageConverter.setTypeIdPropertyName("_typeId");

  Map<String, Class<?>> typeIdMappings = new HashMap<String, Class<?>>();
  typeIdMappings.put("order", Order.class);
  messageConverter.setTypeIdMappings(typeIdMappings);

  return messageConverter;
}
```

这样的话，消息的_typeId属性中就不用发送全限定类型了，而是会发送order值。在接收端的应用中，将会配置类似的消息转换器，将order映射为它自己能够理解的订单类型。在接收端的订单可能位于不同的包中、有不同的类名，甚至可以只包含发送者Order属性的一个子集。

对消息进行后期处理

假设除了利润丰厚的Web业务之外，Taco Cloud还决定开几家实体的连锁taco店，鉴于任何一家餐馆都可能成为Web业务的运行中心，在餐馆中他们需要有一种方式告诉厨房订单的来源，这样厨房的工作人员就能为店面里的订单和Web上的订单执行不同的流程。

我们尽可以在Order对象上添加一个新的source属性，让它携带该信息：如果是在线订单，就将其设置为WEB；如果是店面里的订单，就将其设置为STORE。但是，这样我们就需要同时修改Web站点的Order类和厨房应用的Order类，但实际上只有taco的准备人员需要该信息。

有一种更简单的方案，就是为消息添加一个自定义的头部，让它携

带订单的来源。如果我们使用send()方法来发送taco订单，那么通过调用Message对象的setStringProperty()方法非常容易实现：

```
jms.send("tacocloud.order.queue",
    session -> {
        Message message = session.createObjectMessage(order);
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
    });
```

但是，这里的问题在于我们并没有使用send()。在使用convertAndSend()方法的时候，Message是在底层创建的，我们无法访问到它。

幸好，还有一种方式能够在发送之前修改底层创建的Message对象。我们可以传递一个MessagePostProcessor作为convertAndSend()的最后一个参数，借助它我们可以在Message创建之后做任何想做的事情。如下的代码依然使用了convertAndSend()，但是它能够在消息发送之前使用MessagePostProcessor添加X_ORDER_SOURCE头信息：

```
jms.convertAndSend("tacocloud.order.queue", order, new MessagePostProcesso
r() {
  @Override
  public Message postProcessMessage(Message message) throws JMSException {
    message.setStringProperty("X_ORDER_SOURCE", "WEB");
    return message;
  }
});
```

你可能已经发现MessagePostProcessor是一个函数式接口。这意味着我们可以将匿名内部类替换为lambda，进一步简化它：

```
jms.convertAndSend("tacocloud.order.queue", order,
    message -> {
```

```
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
        return message;
    });
```

尽管在这里我们只是将这个特殊的MessagePostProcessor用到了本次convertAndSend()方法调用中，但是你可能会发现在你的代码中会在不同的地方多次调用convertAndSend()，它们均会用到相同的MessagePostProcessor。在这种情况下，方法引用是比lambda更好的方案，它能避免不必要的代码重复：

```
@GetMapping("/convertAndSend/order")
public String convertAndSendOrder() {
  Order order = buildOrder();
  jms.convertAndSend("tacocloud.order.queue", order,
      this::addOrderSource);
  return "Convert and sent order";
}

private Message addOrderSource(Message message) throws JMSException {
  message.setStringProperty("X_ORDER_SOURCE", "WEB");
  return message;
}
```

我们已经看到了多种发送消息的方式，但是如果消息无人接收，那么只发送消息也没什么价值。接下来，我们看一下如何使用Spring和JMS接收消息。

8.1.3 接收JMS消息

在消费消息的时候，我们可以选择拉取模式（pull model）和推送模式（push model），前者会在我们的代码中请求消息并一直等待直到消息到达为止，而后者则会在消息可用的时候自动在你的代码中执行。

JmsTemplate提供了多种方式来接收消息，但它们使用的都是拉取模式。我们可以调用其中的某个方法来请求消息，而线程会一直阻塞到一个消息抵达为止（这可能马上发生，也可能需要等待一会儿）。

另外，我们也可以使用推送模式，在这种情况下，我们会定义一个消息监听器，每当有消息可用时，它就会被调用。

这两种方案能够适用于各种用户场景。人们普遍觉得推送模式是更好的方案，因为它不会阻塞线程；但是，在某些场景下，如果消息抵达的速度太快，那么监听器可能会过载。而拉取模式允许消费者声明它们何时才为接收新消息做好准备。

我们将会看一下这两种方案，首先从JmsTemplate提供的拉取模式开始。

使用**JmsTemplate**来接收消息

JmsTemplate提供了多个对代理的拉取方法，其中包括：

```
Message receive() throws JmsException;
Message receive(Destination destination) throws JmsException;
Message receive(String destinationName) throws JmsException;

Object receiveAndConvert() throws JmsException;
Object receiveAndConvert(Destination destination) throws JmsException;
Object receiveAndConvert(String destinationName) throws JmsException;
```

我们可以看到，这6个方法简直就是JmsTemplate中send()和convertAndSend()方法的镜像。receive()方法接收原始的Message，而receiveAndConvert()则会使用一个配置好的消息转换器将消息转换成领

域对象。对于其中的每种方法，我们都可以指定Destination或者包含目的地名称的String值，否则，我们将会从默认目的地拉取消息。

为了实际看一下它是如何运行的，我们编写代码从tacocloud.order.queue目的地拉取一个Order。程序清单8.2展现了OrderReceiver，这个服务组件会使用JmsTemplate.receive()来接收订单数据。

程序清单8.2　从队列拉取订单

```
package tacos.kitchen.messaging.jms;
import javax.jms.Message;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.stereotype.Component;

@Component
public class JmsOrderReceiver implements OrderReceiver {
  private JmsTemplate jms;
  private MessageConverter converter;

  @Autowired
  public JmsOrderReceiver(JmsTemplate jms, MessageConverter converter) {
    this.jms = jms;
    this.converter = converter;
  }
  public Order receiveOrder() {
    Message message = jms.receive("tacocloud.order.queue");
    return (Order) converter.fromMessage(message);
  }
}
```

这里我们使用String值来指定从哪个目的地拉取订单。receive()返回的是没有经过转换的Message，但是，我们真正需要的是Message中的Order，所以接下来我们要做的事情就是使用被注入的消息转换器对消

息进行转换。消息中的type ID属性将会指导转换器将消息转换成Order，但它返回的是Object，所以在最终返回之前要进行类型转换。

如果我们要探查消息的属性和消息头信息，那么接收原始的Message对象可能会非常有用。但是，通常来讲，我们只需要消息的载荷。将载荷转换成领域对象是一个需要两步操作的过程，而且它需要将消息转换器注入组件中。如果你只关心载荷，那么使用receiveAndConvert()会更简单一些。程序清单 8.3 展现了如何使用receiveAndConvert()替换receive()来重新实现JmsOrderReceiver。

程序清单8.3　接收已经转换好的Order对象

```
package tacos.kitchen.messaging.jms;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class JmsOrderReceiver implements OrderReceiver {
  private JmsTemplate jms;

  @Autowired
  public JmsOrderReceiver(JmsTemplate jms) {
    this.jms = jms;
  }

  public Order receiveOrder() {
    return (Order) jms.receiveAndConvert("tacocloud.order.queue");
  }
}
```

这个新版本的JmsOrderReceiver的receieveOrder()被简化到了只有一行代码。同时，我们不再需要将MessageConverter注入进来了，因为所有的操作都会在receiveAndConvert()方法的幕后完成。

在继续学习下面的内容之前，我们考虑一下如何在Taco Cloud厨房应用中使用receiveOrder()。Taco Cloud厨房中的厨师可能会按下一个按钮或者采取其他操作，表明他已经准备好开始做taco了。此时，receiveOrder()会被调用，然后对receive()或receiveAndConvert()的调用将会阻塞。在订单消息抵达之前，这里不会发生任何事情。一旦订单抵达，对receiveOrder()的调用将会把该订单信息返回，订单的详细信息会展现给厨师，这样他就可以开始工作了。对于拉取模式来说，这似乎是一种很自然的选择。

接下来，我们看一下如何通过声明JMS监听器来实现推送模式。

声明消息监听器

拉取模式需要显式调用receive()或receiveAndConvert()才能接收消息，与之不同，消息监听器是一个被动的组件，在消息抵达之前，它会一直处于空闲状态。

要创建能够对JMS消息做出反应的消息监听器，我们需要为组件中的某个方法添加@JmsListener注解。程序清单8.4展示了一个新的OrderListener组件，它会被动地监听消息，而不是主动请求消息。

程序清单8.4　监听订单消息的OrderListener组件

```
package tacos.kitchen.messaging.jms.listener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
```

```
public class OrderListener {
  private KitchenUI ui;

  @Autowired
  public OrderListener(KitchenUI ui) {
    this.ui = ui;
  }

  @JmsListener(destination = "tacocloud.order.queue")
  public void receiveOrder(Order order) {
    ui.displayOrder(order);
  }
}
```

receiveOrder()方法使用了JmsListener注解，这样它就会监听tacocloud.order.queue目的地的消息。该方法不需要使用JmsTemplate，也不会被你的应用显式调用。相反，Spring中的框架代码会等待消息抵达指定的目的地，当消息到达时，receiveOrder()方法会被自动调用，并且会将消息中的Order载荷作为参数。

从很多方面来讲，@JmsListener注解都和Spring MVC中的请求映射注解很相似，比如@GetMapping或@PostMapping。在Spring MVC中，带有请求映射注解的方法会响应指定路径的请求。与之类似，使用@JmsListener注解的方法会对到达指定目的地的消息做出响应。

消息监听器通常被视为最佳选择，因为它不会导致阻塞，并且能够快速处理多个消息。但是在Taco Cloud中，它可能并不是最佳的方案。在系统中，厨师是一个重要的瓶颈，他们可能无法在接收到订单的时候立即准备taco。当新订单出现在屏幕上的时候，可能上一个订单刚刚完成一半。厨房用户界面需要在订单到达时进行缓冲，避免给厨房人员带来过重的负载。

这并不是说消息监听器不好。相反，如果消息能够快速得到处理，那么它们是非常适合的方案。但是，如果消息处理器需要根据自己的时间请求更多消息，那么JmsTemplate提供的拉取模式会更加合适。

JMS是由标准Java规范定义的，所以它得到了众多代理实现的支持，在Java中实现消息时它是常见的可选方案。但是JMS有一些缺点，尤其是作为Java规范，它只能用在Java应用中。RabbitMQ和Kafka等较新的消息传递方案克服了这些缺点，可以用于JVM之外的其他语言和平台。让我们把JMS放在一边，看看如何使用RabbitMQ实现taco订单的消息传递。

8.2 使用RabbitMQ和AMQP

RabbitMQ可以说是AMQP最杰出的实现，它提供了比JMS更高级的消息路由策略。JMS消息使用目的地名称来寻址，接收者要从这里检索消息，而AMQP消息使用Exchange和routing key来寻址，这样消息就与接收者要监听的队列解耦了。Exchange和队列的关系如图8.1所示。

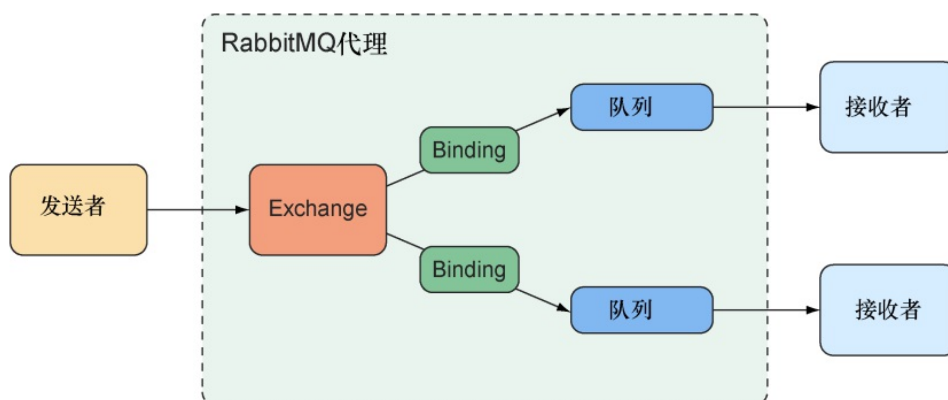


图8.1 发送到RabbitMQ Exchange的消息会基于routing key和binding被路由到一个或多个队列上

当消息抵达RabbitMQ代理的时候，它会进入为其设置的Exchange上。Exchange负责将它路由到一个或多个队列中，这个过程会根据Exchange的类型、Exchange和队列之间的binding以及消息的routing key进行路由。

这方面有多个不同类型的Exchange，包括以下内容。

- Default：这是代理创建的特殊Exchange。它会将消息路由至名字与消息routing key相同的队列。所有的队列都会自动绑定至Default Exchange。
- Direct：如果消息的routing key与队列的binding key相同，那么消息将会路由到该队列上。
- Topic：如果消息的routing key与队列binding key（可能会包含通配符）匹配，那么消息将会路由到一个或多个这样的队列上。
- Fanout：不管routing key和binding key是什么，消息都将会路由到所有绑定队列上。
- Headers：与Topic Exchange类似，只不过要基于消息的头信息进行路由，而不是routing key。
- Dead letter：捕获所有无法投递（也就是它们无法匹配所有已定义的Exchange和队列的binding关系）的消息。

最简单的Exchange形式是Default和Fanout，因为它们大致对应了JMS中的队列和主题，但是其他的Exchange允许我们定义更加灵活的路由模式。

这里最重要的是要明白消息会通过routing key发送至Exchange，而消息要在队列中被消费。它们如何从Exchange路由至队列取决于binding

的定义以及哪种方式最适合我们的使用场景。

至于使用哪种Exchange类型以及如何定义从Exchange到队列的binding，这本身与如何在Spring应用中发送和接收消息关系不大。因此，我们更加关心如何编写使用Rabbit发送和接收消息的代码。

> 注意：关于如何绑定队列到Exchange的更详细讨论，请参考Alvaro Videla和Jason J.W. Williams编写的*RabbitMQ in Action*（Manning，2012）。

8.2.1　添加RabbitMQ到Spring中

在使用Spring发送和接收RabbitMQ消息之前，我们需要将Spring Boot的AMQP starter依赖添加到构建文件中，替换上文中Artemis或ActiveMQ starter的位置：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

添加AMQP starter到构建文件中之后，将会触发自动配置功能，这样会为我们创建一个AMQP连接工厂和RabbitTemplate bean，以及其他的一些支撑组件。我们要使用Spring发送和接收RabbitMQ代理的消息，只需要添加这项依赖就可以了。但是，这里还有一些我们需要掌握的有

用属性，如表8.4所示。

表**8.4**　配置**RabbitMQ**代理位置和凭证的属性

属性	描述
spring.rabbitmq.addresses	逗号分隔的RabbitMQ代理地址列表
spring.rabbitmq.host	代理的主机（默认为localhost）
spring.rabbitmq.port	代理的端口（默认为5672）
spring.rabbitmq.username	访问代理所使用的用户名（可选）
spring.rabbitmq.password	访问代理所使用的密码（可选）

对于开发来说，我们可能会使用不需要认证的RabbitMQ代理，它运行在本地机器上并监听5672端口。在开发阶段，这些属性可能没有太大的用处，但是当应用程序投入生产环境时，它们无疑是非常有用的。

假设我们要将应用投入生产环境，RabbitMQ代理位于名为rabbit.tacocloud.com服务器上，监听5673端口并且需要认证。在这种情况下，当prod profile处于激活状态时，application.yml文件中的如下配置将会设置这些属性：

```
spring:
  profiles: prod
```

```
  rabbitmq:
    host: rabbit.tacocloud.com
    port: 5673
    username: tacoweb
    password: l3tm31n
```

在我们的应用中，RabbitMQ已经配置好了，接下来就可以使用RabbitTemplate发送消息了。

8.2.2 通过RabbitTemplate发送消息

Spring 对 RabbitMQ 消息支持的核心是 RabbitTemplate。RabbitTemplate 与JmsTemplate类似，提供了一组相似的方法。但是，我们将会看到，这里有一些细微的差异，这是与RabbitMQ独特的运行方式有关的。

在使用RabbitTemplate发送消息方面，我们可以使用与JmsTemplate中同名的send()和convertAndSend()方法。但是，与JmsTemplate的方法只是将消息路由至队列或主题不同，RabbitTemplate会按照Exchanges和routing key来发送消息。下面列出关于使用RabbitTemplate发送消息比较重要的一些方法[1]：

```
// 发送原始的消息
void send(Message message) throws AmqpException;
void send(String routingKey, Message message) throws AmqpException;
void send(String exchange, String routingKey, Message message)
                    throws AmqpException;

// 发送根据对象转换而成的消息
void convertAndSend(Object message) throws AmqpException;
void convertAndSend(String routingKey, Object message)
                    throws AmqpException;
void convertAndSend(String exchange, String routingKey,
```

```
                    Object message) throws AmqpException;

// 发送根据对象转换而成的消息并且带有后期处理的功能
void convertAndSend(Object message, MessagePostProcessor mPP)
                    throws AmqpException;
void convertAndSend(String routingKey, Object message,
                    MessagePostProcessor messagePostProcessor)
                    throws AmqpException;
void convertAndSend(String exchange, String routingKey,
                    Object message,
                    MessagePostProcessor messagePostProcessor)
                    throws AmqpException;
```

我们可以看到，这些方法与JmsTemplate中对应的方法遵循了相同的模式。前3个send()方法都是发送原始的Message对象。接下来的3个convertAndSend()方法会接受一个对象，这个对象会在发送之前在幕后转换成Message。最后的3个convertAndSend()方法与前面的3个方法类似，但是它们还会接受一个MessagePostProcessor对象，这个对象能够在Message发送至代理之前对其进行操作。

这些方法与JmsTemplate对应方法的不同之处在于，它们会接受String类型的值以指定Exchange和routing key，而不像JmsTemplate那样接受目的地名称（或Destination）。没有接受Exchange参数的方法会将消息发送至Default Exchange。与之类似，没有指定routing key的方法会把消息路由至默认的routing key。

接下来，我们看一下如何使用RabbitTemplate发送taco订单。有一种方式是使用send()方法，如程序清单8.5所示。但是，在调用send()之前，我们需要将Order对象转换为Message。RabbitTemplate能够通过getMessageConverter()方法获取消息转换器，否则，这项工作会非常乏味。

程序清单8.5　使用RabbitTemplate.send()发送消息

```
package tacos.messaging;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import tacos.Order;

@Service
public class RabbitOrderMessagingService
       implements OrderMessagingService {
  private RabbitTemplate rabbit;

  @Autowired
  public RabbitOrderMessagingService(RabbitTemplate rabbit) {
    this.rabbit = rabbit;
  }

  public void sendOrder(Order order) {
    MessageConverter converter = rabbit.getMessageConverter();
    MessageProperties props = new MessageProperties();
    Message message = converter.toMessage(order, props);
    rabbit.send("tacocloud.order", message);
  }
}
```

有了MessageConverter之后，将Order转换成Message就是非常简单的任务了。我们必须通过MessageProperties来提供消息属性，但是如果我们不需要设置任何这样的属性，使用默认的MessageProperties实例就可以了。随后，剩下的就是调用send()了，并将Exchange和routing key（这两者都是可选的）连同消息一起传递过去。在本例中，我们只指定了routing key（tacocloud.order）和消息本身，所以会使用默认的Exchange。

这里提到了默认的Exchange，它的名字是“”（空的String），对应

RabbitMQ代理自动生成的Default Exchange。与之相似，默认的routing key也是“”（它的路由将会取决于Exchange以及相应的binding）。我们可以通过设置spring.rabbitmq.template.exchange和spring.rabbitmq.template.routing-key属性重写这些默认值：

```
spring:
  rabbitmq:
    template:
      exchange: tacocloud.orders
      routing-key: kitchens.central
```

在本例中，所有未指明Exchange的消息都将会自动发送至名为tacocloud.orders的Exchange。如果在调用send()或convertAndSend()的时候也没有指定routing key，那么消息将会使用值为kitchens.central的routing key。

通过消息转换器创建Message对象是非常简单的，但是使用convertAndSend()让RabbitTemplate处理所有的转换操作则会更加简单：

```
public void sendOrder(Order order) {
  rabbit.convertAndSend("tacocloud.order", order);
}
```

配置消息转换器

默认情况下，消息转换是通过SimpleMessageConverter来实现的，它能够将简单类型（如String）和Serializable对象转换成Message对象。但是，Spring为RabbitTemplate提供了多个消息转换器，包括下面内容。

- Jackson2JsonMessageConverter：使用Jackson 2 JSON实现对象和

JSON的相互转换。

- MarshallingMessageConverter：使用Spring的Marshaller和Unmarshaller进行转换。
- SerializerMessageConverter：使用Spring的Serializer和Deserializer转换String和任意种类的原生对象。
- SimpleMessageConverter：转换String、字节数组和Serializable类型。
- ContentTypeDelegatingMessageConverter：基于contentType头信息，将转换功能委托给另外一个MessageConverter。
- MessagingMessageConverter：将消息转换功能委托给另外一个MessageConverter，并将头信息的转换委托给AmqpHeaderConverter。

如果需要变更消息转换器，就要配置一个类型为MessageConverter的bean。例如，对于基于JSON的转换，我们可以按照如下的方式来配置Jackson2JsonMessageConverter：

```
@Bean
public MessageConverter messageConverter() {
  return new Jackson2JsonMessageConverter();
}
```

Spring Boot的自动配置功能会发现这个bean，并将它注入RabbitTemplate中，替换默认的消息转换器。

设置消息属性

与在JMS中一样，我们可能需要在发送的消息中添加一些头信息。例如，假设我们需要为所有通过Taco Cloud Web站点提交的订单添加一

个X_ORDER_SOURCE信息。在自行创建Message的时候，我们可以通过MessageProperties实例设置头信息，随后将这个对象传递给消息转换器。回到程序清单8.5的sendOrder()方法，我们需要做的就是添加一行设置头信息的代码：

```
public void sendOrder(Order order) {
  MessageConverter converter = rabbit.getMessageConverter();
  MessageProperties props = new MessageProperties();
  props.setHeader("X_ORDER_SOURCE", "WEB");
  Message message = converter.toMessage(order, props);
  rabbit.send("tacocloud.order", message);
}
```

但是，在使用convertAndSend()的时候，我们无法快速访问MessageProperties对象。不过，此时MessagePostProcessor可以帮助我们：

```
@Override
public void sendOrder(Order order) {
  rabbit.convertAndSend("tacocloud.order.queue", order,
      new MessagePostProcessor() {
        @Override
        public Message postProcessMessage(Message message)
            throws AmqpException {
          MessageProperties props = message.getMessageProperties();
          props.setHeader("X_ORDER_SOURCE", "WEB");
          return message;
        }
  });
}
```

在这里，我们为convertAndSend()提供了MessagePostProcessor接口的匿名内部类实现。在postProcessMessage()中，我们从Message中拉取MessageProperties对象，然后通过setHeader()方法设置X_ORDER_SOURCE头信息。

现在，我们已经看到了如何通过RabbitTemplate发送消息，接下来我们转换视角看一下如何接收来自RabbitMQ队列的消息。

8.2.3 接收来自RabbitMQ的消息

我们看到使用RabbitTemplate发送消息与使用JmsTemplate发送消息并没有太大差别。实际上，接收来自RabbitMQ队列的消息也与JMS没有太大差别。与JMS类似，我们有两个可选方案。

- 使用RabbitTemplate从队列拉取消息。
- 将消息推送至带有@RabbitListener注解的方法。

我们首先看一下基于拉取的RabbitTemplate.receive()方法。

使用**RabbitTemplate**接收消息

RabbitTemplate提供了多个从队列拉取消息的方法。其中，最有用的方法如下所示：

```
// 接收消息
Message receive() throws AmqpException;
Message receive(String queueName) throws AmqpException;
Message receive(long timeoutMillis) throws AmqpException;
Message receive(String queueName, long timeoutMillis) throws AmqpException
;

// 接收由消息转换而成的对象
Object receiveAndConvert() throws AmqpException;
Object receiveAndConvert(String queueName) throws AmqpException;
Object receiveAndConvert(long timeoutMillis) throws AmqpException;
Object receiveAndConvert(String queueName, long timeoutMillis) throws
     AmqpException;
```

```
// 接收由消息转换而成的类型安全的对象
<T> T receiveAndConvert(ParameterizedTypeReference<T> type) throws
     AmqpException;
<T> T receiveAndConvert(String queueName, ParameterizedTypeReference<T> ty
pe)
     throws AmqpException;
<T> T receiveAndConvert(long timeoutMillis, ParameterizedTypeReference<T>
     type) throws AmqpException;
<T> T receiveAndConvert(String queueName, long timeoutMillis,
     ParameterizedTypeReference<T> type)
    throws AmqpException;
```

这些方法对应于前文所述的send()和convertAndSend()方法。send()用于发送原始的Message对象，而receive()则会接收来自队列的原始Message对象。与之类似，receiveAndConvert()接收消息并且在返回之前使用一个消息转换器将它们转换为领域对象。

但是，在方法签名上有一些明显的不同。首先，这些方法都不会接收Exchange 和routing key作为参数。这是因为Exchange和routing key是用来将消息路由至队列的，在消息位于队列中之后，它们的目的地是将它们从队列中拉取下来的消费者。消费消息的应用本身并不需要关心Exchange和routing key。消费消息的应用只需要知道队列信息就可以了。

你可能会注意到，很多方法都接收一个long类型的参数，用来指定接收消息的超时时间。默认情况下，接收消息的超时时间是0毫秒。也就是说，调用receive()会立即返回，如果没有可用消息，那么返回值是null。这是与JmsTemplate的receive()的一个显著差异。通过传入一个超时时间的值，我们就可以让receive()和receiveAndConvert()阻塞，直到消息抵达或者超时时间过期。但是，即便我们设置了非零的超时时间，在

代码中依然要处理null返回值的场景。

接下来，我们看一下如何实际使用它们。程序清单8.6展现了一个新的基于Rabbit的OrderReceiver实现，它使用RabbitTemplate来接收订单。

程序清单8.6　通过RabbitTemplate从RabbitMQ中拉取订单

```
package tacos.kitchen.messaging.rabbit;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class RabbitOrderReceiver {
  private RabbitTemplate rabbit;
  private MessageConverter converter;

  @Autowired
  public RabbitOrderReceiver(RabbitTemplate rabbit) {
    this.rabbit = rabbit;
    this.converter = rabbit.getMessageConverter();
  }

  public Order receiveOrder() {
    Message message = rabbit.receive("tacocloud.orders");
    return message != null
           ? (Order) converter.fromMessage(message)
           : null;
  }
}
```

所有的操作都发生在receiveOrder()方法中。它调用了注入的RabbitTemplate对象的receive()方法，从名为tacocloud.orders的队列中拉取一个订单。它并没有提供超时值，所以我们只能假定这个调用会马上返回，要么得到Message对象，要么返回null。如果返回Message对象，

就使用RabbitTemplate中的MessageConverter将Message转换成一个Order对象。如果receive()方法返回null，我们就将null作为返回值。

根据使用场景，我们也许能够容忍一定的延迟。例如，在Taco Cloud厨房悬挂的显示器中，如果没有订单，我们可以稍等一会儿。假设在放弃之前，我们决定等待30秒钟，那么receiveOrder()方法可以修改为传递30 000毫秒的延迟给receive()方法：

```
public Order receiveOrder() {
  Message message = rabbit.receive("tacocloud.order.queue", 30000);
  return message != null
         ? (Order) converter.fromMessage(message)
         : null;
}
```

如果你像我一样，觉得使用这样一个硬编码的数字会让人觉得不舒服，那么你可能会想，创建一个带有@ConfigurationProperties注解的类并使用Spring Boot的配置属性来设置超时时间，可能会是更好的方案。在一点上，我的想法和你一样，只不过Spring Boot已经为我们提供了一个这样的配置属性。如果你想要通过配置来设置超时时间，只需要在调用receive()的时候移除超时值，并将超时时间设置为spring.rabbitmq.template.receive-timeout属性即可：

```
spring:
  rabbitmq:
    template:
      receive-timeout: 30000
```

回到receiveOrder()方法，我们必须要使用RabbitTemplate中的消息转换器才能将传入的Message对象转换成Order对象。但是，既然

RabbitTemplate已经携带了消息转换器，它为什么不能自动为我们进行转换呢？这就是receiveAndConvert()方法所做的事情。借助receiveAndConvert()，我们可以将receiveOrder()重写为：

```
public Order receiveOrder() {
  return (Order) rabbit.receiveAndConvert("tacocloud.order.queue");
}
```

看起来简单了许多，对吧。唯一让我觉得麻烦的就是从Object到Order的类型转换。不过，这种转换还有另外一种实现方式。我们可以传递一个ParameterizedTypeReference引用给receiveAndConvert()，这样我们就可以直接得到Order对象了：

```
public Order receiveOrder() {
  return rabbit.receiveAndConvert("tacocloud.order.queue",
                  new ParameterizedTypeReference<Order>() {});
}
```

关于这种方式是否真的比类型转换更好，依然还有争论，但是它确实能够更加确保类型安全。唯一需要注意的是，要在receiveAndConvert()中使用ParameterizedTypeReference，消息转换器必须要实现SmartMessageConverter，目前Jackson2JsonMessageConverter是唯一一个可选的内置实现。

RabbitTemplate提供的拉取模式适用于很多使用场景，但是有时候监听消息并在消息抵达的时候对其进行处理会更好一些。接下来，我们看一下如何编写消息驱动的bean，让它对RabbitMQ消息做出回应。

使用监听器处理**RabbitMQ**的消息

Spring提供了RabbitListener实现消息驱动的RabbitMQ bean，对应于JmsListener。为了声明当消息抵达RabbitMQ队列时某个方法应该被调用，我们可以为bean的方法添加@RabbitListener注解。

例如，程序清单8.7展现了OrderReceiver的RabbitMQ实现，它通过注解声明要监听订单消息，而不是使用RabbitTemplate进行轮询。

程序清单8.7　将方法声明为RabbitMQ的消息监听器

```
package tacos.kitchen.messaging.rabbit.listener;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class OrderListener {
  private KitchenUI ui;

  @Autowired
  public OrderListener(KitchenUI ui) {
    this.ui = ui;
  }

  @RabbitListener(queues = "tacocloud.order.queue")
  public void receiveOrder(Order order) {
    ui.displayOrder(order);
  }
}
```

你肯定会发现它与程序清单8.4的代码非常相似。确实，唯一的变更就是监听器的注解，从@JmsListener换成了@RabbitListener。尽管@RabbitListener注解非常棒，但是几乎重复的代码让我无法找出@RabbitListener具有什么在@JmsListener中没有提到的功能。当消息从各自的代理推送过来的时候，这两个注解都非常适合用来编写对应的代码，其中@JmsListener对应的是JMS代理，而@RabbitListener对应的是

RabbitMQ代理。

在前面的段落中，你可能会对@RabbitListener感到索然无趣，但是这并非我的本意。实际上，@RabbitListener和@JmsListener的运行方式非常相似是一件令人兴奋的事情。这意味着当我们使用RabbitMQ替代Artemis或ActiveMQ的时候，不需要学习全新的编程模型。同样令人兴奋的是，RabbitTemplate和JmsTemplate之间也具有这样的相似性。

让我们暂且保持一下这种兴奋，在本章结束之前，我们看一下Spring支持的另外一个消息方案：Apache Kafka。

8.3 使用Kafka的消息

Apache Kafka是我们在本章研究的最新的消息方案。乍看上去，Kafka是与ActiveMQ、Artemis或Rabbit类似的消息代理，其实Kafka有一些独特的技巧。

Kafka设计为集群运行，从而能够实现很强的可扩展性。通过将主题在集群的所有实例上进行分区（partition），它能够具有更强的弹性。RabbitMQ主要处理Exchange中的队列，而Kafka仅使用主题实现消息的发布/订阅。

Kafka主题会复制到集群的所有代理上。集群中的每个节点都会担任一个或多个主题的首领（leader），负责该主题的数据并将其复制到集群中的其他节点上。

更进一步来讲，每个主题可以划分为多个分区。在这种情况下，集群中的每个节点是某个主题一个或多个分区的首领，但并不是整个主题的首领。主题的责任会在所有节点间进行拆分。图8.2阐述了它是如何运行的。

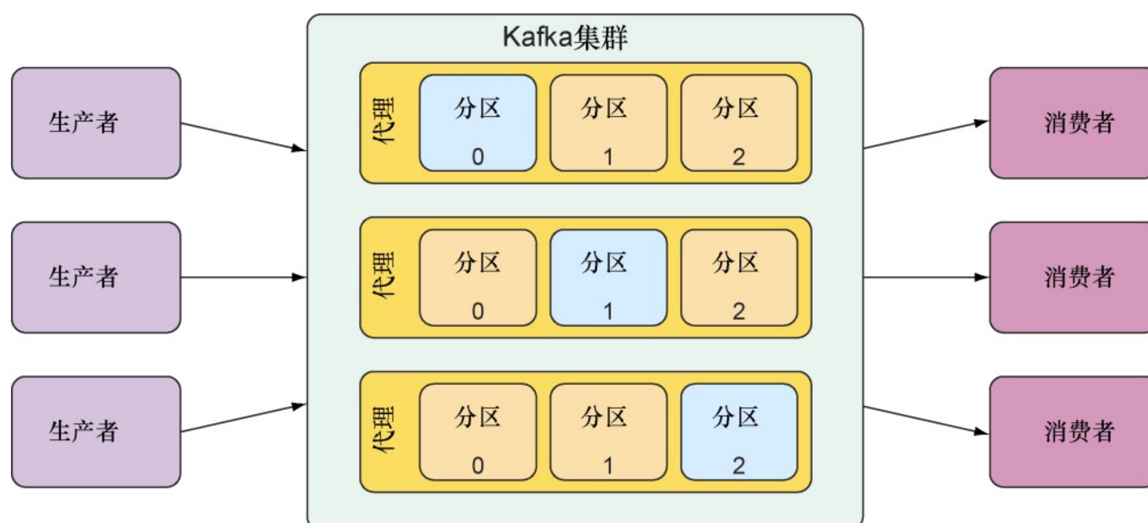


图8.2　Kafka集群是由多个代理组成的，每个代理作为主题分区的首领

关于Kafka的独特架构，我建议你阅读Dylan Scott编写的*Kafka in Action*（Manning，2017）。就我们来讲，我们将会关注如何通过Spring发送和接收Kafka的消息。

8.3.1　为Spring搭建支持Kafka消息的环境

为了搭建Kafka的消息环境，我们需要添加对应的依赖到构建文件中。但是，与JMS和RabbitMQ方案不同，并没有针对Kafka的Spring Boot starter。不过，不用担心，我们只需要添加一项依赖：

```
<dependency>
```

```
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

这项依赖会为我们的项目引入Kafka所需的所有内容。另外，它的出现会触发Spring Boot对Kafka的自动配置，除了其他功能之外，它会在Spring应用上下文中创建一个KafkaTemplate。我们所需要做的就是注入KafkaTemplate并使用它来发布和接收消息。

但是，在发送和接收消息之前，我们还需要注意使用Kafka时的一些属性。具体来讲，KafkaTemplate默认会使用localhost上监听9092端口的Kafka代理。在开发应用的时候，在本地启动Kafka代理没有问题，但是在投入生产的时候，我们需要配置不同的主机和端口。

spring.kafka.bootstrap-servers属性能够设置一个或多个Kafka服务器的地址，系统将会使用它来建立到Kafka集群的初始连接。例如，集群中的某个服务器运行在kafka.tacocloud.com上并监听9092端口，那么我们可以按照如下的方式在YAML中配置它的位置：

```
spring:
  kafka:
    bootstrap-servers:
    - kafka.tacocloud.com:9092
```

但是需要注意spring.kafka.bootstrap-servers是复数形式，它能接受一个列表。所以，我们可以提供集群中的多个Kafka服务器：

```
spring:
  kafka:
    bootstrap-servers:
    - kafka.tacocloud.com:9092
    - kafka.tacocloud.com:9093
```

```
    - kafka.tacocloud.com:9094
```

Kafka在项目中准备就绪之后，我们就可以发送和接收消息了。我们首先使用KafkaTemplate发送Order对象到Kafka中。

8.3.2 通过KafkaTemplate发送消息

在很多方面，KafkaTemplate与JMS和RabbitMQ对应的模板非常相似。但同时，它也有很大的差异。在发送消息的时候，这一点非常明显：

```
ListenableFuture<SendResult<K, V>> send(String topic, V data);
ListenableFuture<SendResult<K, V>> send(String topic, K key, V data);
ListenableFuture<SendResult<K, V>> send(String topic,
                    Integer partition, K key, V data);
ListenableFuture<SendResult<K, V>> send(String topic,
                    Integer partition, Long timestamp, K key, V data);
ListenableFuture<SendResult<K, V>> send(ProducerRecord<K, V> record);
ListenableFuture<SendResult<K, V>> send(Message<?> message);

ListenableFuture<SendResult<K, V>> sendDefault(V data);
ListenableFuture<SendResult<K, V>> sendDefault(K key, V data);
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition,
                                               K key, V data);
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition,
                                     Long timestamp, K key, V data);
```

我们首先可能会发现，这里没有convertAndSend()方法了。这是因为，KafkaTemplate是通过泛型类型化的，在发送消息的时候，它能够直接处理领域类型。这样的话，所有的send()方法都完成了convertAndSend()的任务。

你可能也会发现，send()和sendDefault()的参数与JMS和Rabbit有很

大的差异。在使用Kafka发送消息的时候，我们可以使用如下参数设置消息该如何进行发送：

- 消息要发送到的主题（send()方法的必选参数）；
- 主题要写入的分区（可选）；
- 记录上要发送的key（可选）；
- 时间戳（可选，默认为System.currentTimeMillis()）；
- 载荷（必选）。

主题和载荷是其中最重要的两个参数。分区和key对于如何使用KafkaTemplate几乎没有影响，只是作为额外的信息提供给send()和sendDefault()。对于我们的场景来说，我们只关心将消息载荷发送到给定的主题，不用担心分区和key的问题。

对于send()方法来说，我们还可以选择发送一个ProducerRecord对象，它只是一个简单类型，将上述的参数放到了一个对象中。我们还可以发送Message对象，但是需要将领域对象转换成Message对象。相对创建和发送ProducerRecord和Message对象，使用其他的方法会更简单一些。

借助KafkaTemplate及其send()方法，我们可以编写一个基于Kafka实现的OrderMessagingService实现。程序清单8.8展现了该实现类。

程序清单8.8　使用KafkaTemplate发送订单

```
package tacos.messaging;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;
```

```
@Service
public class KafkaOrderMessagingService
                                  implements OrderMessagingService {

  private KafkaTemplate<String, Order> kafkaTemplate;

  @Autowired
  public KafkaOrderMessagingService(
          KafkaTemplate<String, Order> kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
  }

  @Override
  public void sendOrder(Order order) {
    kafkaTemplate.send("tacocloud.orders.topic", order);
  }

}
```

在这个OrderMessagingService的新实现中，sendOrder()用到了注入的KafkaTemplate对象的send()方法，将Order发送到名为tacocloud.orders.topic的主题中。除了代码中随处可见的“Kafka”之外，它其实与为JMS和Rabbit编写的代码并没有太大的差异。

如果你想要设置默认主题，那么可以稍微简化一下sendOrder()。首先，通过spring.kafka.template.default-topic属性，我们可以将默认主题设置为tacocloud.orders.topic：

```
spring:
  kafka:
    template:
      default-topic: tacocloud.orders.topic
```

然后，在sendOrder()方法中，我们就可以调用sendDefault()而不是send()了，这样可以不用指定主题的名称：

```
@Override
```

```
public void sendOrder(Order order) {
  kafkaTemplate.sendDefault(order);
}
```

现在，我们已经编写完发送消息的代码了。接下来，我们转移一下注意力，编写从Kafka中接收消息的代码。

8.3.3　编写Kafka监听器

除了send()和sendDefault()特有的方法签名之外，KafkaTemplate与JmsTemplate和RabbitTemplate另一个不同之处在于它没有提供接收消息的方法。这意味着在Spring中想要消费来自Kafka主题的消息只有一种办法，就是编写消息监听器。

对于Kafka消息来说，消息监听器是通过带有@KafkaListener注解的方法来实现的。@KafkaListener大致对应于@JmsListener和@RabbitListener，并且使用方式也基本相同。如下的程序清单展示了为Kafka编写的基于监听器的订单接收器。

程序清单8.9　使用@KafkaListener接收订单

```
package tacos.kitchen.messaging.kafka.listener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;
import tacos.Order;
import tacos.kitchen.KitchenUI;
@Component
public class OrderListener {

  private KitchenUI ui;

  @Autowired
```

```
  public OrderListener(KitchenUI ui) {
    this.ui = ui;
  }

  @KafkaListener(topics="tacocloud.orders.topic")
  public void handle(Order order) {
    ui.displayOrder(order);
  }

}
```

handle()方法使用了@KafkaListener注解，表明当有消息抵达名为tacocloud.orders.topic的主题时，该方法将会被调用。如程序清单8.9所示，我们只将Order（载荷）对象传递给了handle()。如果你想要获取消息中其他的元数据，我们也可以接受ConsumerRecord或Message对象。

例如，如下的handle()实现接受一个ConsumerRecord，这样我们就能在日志中将消息的分区和时间戳记录下来：

```
@KafkaListener(topics="tacocloud.orders.topic")
public void handle(Order order, ConsumerRecord<Order> record) {
  log.info("Received from partition {} with timestamp {}",
      record.partition(), record.timestamp());
  ui.displayOrder(order);
}
```

类似地，我们还可以用一个Message对象来替代ConsumerRecord，并且能够达到相同的目的：

```
@KafkaListener(topics="tacocloud.orders.topic")
public void handle(Order order, Message<Order> message) {
MessageHeaders headers = message.getHeaders();
  log.info("Received from partition {} with timestamp {}",
      headers.get(KafkaHeaders.RECEIVED_PARTITION_ID)
      headers.get(KafkaHeaders.RECEIVED_TIMESTAMP));
  ui.displayOrder(order);
}
```

值得一提的是，消息载荷也可以通过ConsumerRecord.value()或Message.getPayload()获取到。这意味着我们可以通过这些对象获取Order，而不必直接将其作为handle()的参数。

8.4　小结

- 异步消息在要通信的应用程序之间提供了一个中间层，这样能够实现更松散的耦合和更强的可扩展性。
- Spring支持使用JMS、RabbitMQ或Apache Kafka实现异步消息。
- 应用程序可以使用基于模板的客户端（JmsTemplate、RabbitTemplate或KafkaTemplate）向消息代理发送消息。
- 接收消息的应用程序可以借助相同的基于模板的客户端以拉取模式消费消息。
- 通过使用消息监听器注解（@JmsListener、@RabbitListener或@KafkaListener），消息也可以推送至消费者的bean方法中。

---

[1]　这些方法是由AmqpTemplate定义的，RabbitTemplate实现了该接口。

第9章　Spring集成

本章内容：

- 实时处理数据
- 定义集成流
- 使用Spring Integration的Java DSL定义
- 与Email、文件系统和其他外部系统进行集成

在旅行时，最让我感到沮丧的一件事就是长途飞行时的互联网连接非常差，或者根本就没有。我喜欢利用空中时间完成一些工作，这本书的很多内容就是这样写出来的。但是，如果没有网络连接，恰好我又想获取某个库或者查看一个JavaDoc文档，那么我就无能为力了。因此，我现在会随身带一本书，以便于在这种场合下阅读。

就像我们需要连接互联网才能提高生产效率一样，很多应用都需要连接外部系统才能完成它们的功能。应用程序可能需要读取或发送Email、与外部API交互或者对写入数据库的数据做出反应。而且，由于

数据是在外部系统读取或写入的，应用可能需要以某种方式处理这些数据，这样才能转换为应用程序自己的领域类。

在本章中，我们将会看到如何使用Spring Integration实现通用的集成模式。Spring Integration是众多集成模式的现成实现，这些模式在Gregor Hohpe和Bobby Woolf编写的《企业集成模式》（*Enterprise Integration Patterns*，Addison-Wesley，2003）中进行了归类。每个模式都实现为一个组件，消息会通过该组件在管道中传递数据。借助Spring配置，我们可以将这些组件组装成一个管道，数据可以通过这个管道来流动。我们从定义一个简单的集成流开始，这个流包含了Spring Integration的众多特性和特点。

9.1 声明一个简单的集成流

通常来讲，在使用Spring Integration创建集成流时，是通过声明一个应用程序能够接收或发送哪些数据到应用程序之外的资源来实现的。应用程序可能集成的资源之一就是文件系统。因此，Spring Integration的很多组件都有读入和写入文件的通道适配器（channel adapter）。

为了熟悉Spring Integration，我们将会创建一个集成流，这个流会写入数据到文件系统中。首先，我们需要添加Spring Integration到项目的构建文件中。对于Maven构建来讲，必要的依赖如下所示：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-integration</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
</dependency>
```

第一项依赖是Spring Integration的Spring Boot starter。不管我们与哪种流进行交互，对于Spring Integration流的开发来讲，这个依赖都是必需的。与所有的Spring Boot starter一样，在Initializr表单中，这个依赖也可以通过复选框进行选择。

第二项依赖是Spring Integration的文件端点模块。这个模块是与外部系统集成的二十多个模块之一。我们会在9.2.9小节中更加详细地讨论端点模块。对于现在来讲，我们只需要知道文件端点模块提供了将文件从文件系统导入集成流和/或将流中的数据写入文件系统的能力即可。

接下来，我们需要为应用创建一种方法，让它能够发送数据到集成流中，这样它才能写入到文件中。为了实现这一点，我们需要创建一个网关接口，这样的网关接口如程序清单9.1所示。

程序清单9.1　将方法调用转换成消息的消息网关接口

```
package sia5;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.integration.file.FileHeaders;
import org.springframework.messaging.handler.annotation.Header;

@MessagingGateway(defaultRequestChannel="textInChannel")        ←--- 声明
消息网关
public interface FileWriterGateway {

  void writeToFile(
      @Header(FileHeaders.FILENAME) String filename,    ←--- 写入文件
      String data);
}
```

尽管这只是一个很简单的Java接口，但是FileWriterGateway有很多东西需要介绍。我们首先看到的是，它使用了@MessagingGateway注解。这个注解会告诉Spring Integration要在运行时生成该接口的实现，这与Spring Data在运行时生成repository接口的实现非常类似。其他地方的代码在希望写入文件的时候将会调用它。

@MessagingGateway的defaultRequestChannel属性表明接口方法调用时所返回的消息要发送至给定的消息通道（message channel）。在本例中，我们声明调用writeToFile()所形成的消息应该发送至名为textInChannel的通道中。

对于writeToFile()方法来说，它以String类型的形式接受一个文件名，另外一个String包含了要写入文件中的文本。关于这个方法的签名，还需要注意filename参数上带有@Header。在本例中，@Header注解表明传递给filename的值应该包含在消息头信息中（通过FileHeaders.FILENAME声明，它将会被解析成file_name），而不是放到消息载荷（payload）中。

现在，我们已经有了消息网关，接下来就需要配置集成流了。尽管我们往构建文件中添加的Spring Integration starter依赖能够启用Spring Integration的自动配置功能，但是满足应用需求的流定义则需要我们自行编写额外的配置。在声明集成流方面，我们有3种配置方案可供选择：

- XML配置；
- Java配置；

- 使用DSL的Java配置。

我们会依次看一下Spring Integration的这 3 种配置风格，首先从较为老式的XML配置开始。

9.1.1 使用XML定义集成流

尽管在本书中，我尽量避免使用XML配置，但是Spring Integration有使用XML定义集成流的漫长历史。所以，我认为至少展现一个XML定义集成流的样例还是很有价值的。程序清单9.2展现了如何使用XML配置示例集成流。

程序清单9.2 使用Spring XML配置定义集成流

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-file="http://www.springframework.org/schema/integration/file"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.x
sd
    http://www.springframework.org/schema/integration/file
    http://www.springframework.org/schema/integration/file/spring-integrat
ion-
     file.xsd">

  <int:channel id="textInChannel" />          ←--- 声明textInChannel

  <int:transformer id="upperCase"
      input-channel="textInChannel"
      output-channel="fileWriterChannel"
      expression="payload.toUpperCase()" />      ←--- 转换文本

  <int:channel id="fileWriterChannel" />             ←--- 声明fileWriterC
```

```
hannel

  <int-file:outbound-channel-adapter id="writer"
    channel="fileWriterChannel"
    directory="/tmp/sia5/files"
    mode="APPEND"
    append-new-line="true" />          ←--- 将文本写入到文件中

</beans>
```

我们将程序清单9.2中的XML拆分讲解一下。

- 我们首先配置了一个名为textInChannel的通道。你会发现，它就是FileWriterGateway的请求通道。当FileWriterGateway的writeToFile()方法被调用的时候，结果形成的消息将会发布到这个通道上。
- 我们还配置了一个转换器（transformer），它会从textInChannel接收消息。它使用Spring表达式语言（Spring Expression Language，SpEL）为消息载荷调用了toUpperCase()方法。进行大写操作之后的结果会发布到fileWriterChannel上。
- 随后，我们配置了名为fileWriterChannel的通道。这个通道会作为一个导线，将转换器与出站通道适配器（outbound channel adapter）连接在一起。
- 最后，我们使用int-file命名空间配置了出站通道适配器。这个XML命名空间是由Spring Integration的文件模块提供的，实现文件写入的功能。按照我们的配置，它从fileWriterChannel接收消息，并将消息的载荷写入到一个文件中，这个文件的名称是由消息头信息中的file_name属性指定的，而存入的目录则是由这里的directory属性指定的。如果文件已经存在，那么文件内容会以新行的方式进行追加，而不是覆盖该文件。

图9.1使用《企业集成模式》中的图形元素样式阐述了这个流。

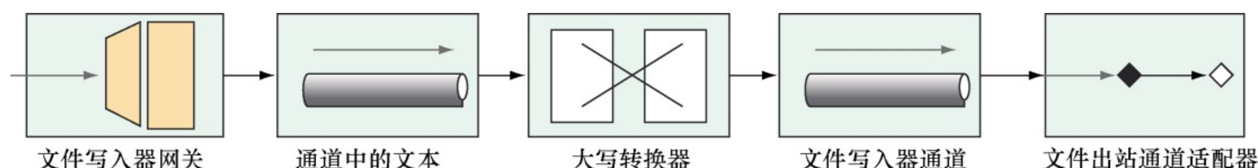


图9.1　文件写入器的集成流

如果想要在Spring Boot应用中使用XML配置，那么我们需要将XML作为资源导入到Spring应用中。最简单的实现方式就是在应用的某个Java配置类上使用Spring的@ImportResource注解：

```
@Configuration
@ImportResource("classpath:/filewriter-config.xml")
public class FileWriterIntegrationConfig { ... }
```

尽管基于XML的配置能够很好地用于Spring Integration，但是大多数的开发人员对于XML的使用越来越谨慎。（正如我所言，在本书中，我会尽量避免使用XML配置。）现在，我们抛开这些尖括号，看一下Spring Integration的Java配置风格。

9.1.2　使用Java配置集成流

大多数的现代Spring应用程序都会避免使用XML配置，而更加青睐Java配置。实际上，在Spring Boot应用中，Java配置是自动化配置功能更自然的补充形式。因此，如果我们要为Spring Boot应用添加集成流，最好使用Java来定义流程。

程序清单9.3展示了使用Java配置编写集成流的一个样例。这里的代码依然是功能相同的文件写入集成流，但是这次我们选择使用Java来实

现。

程序清单9.3　使用Java配置来定义集成流

```
package sia5;
import java.io.File;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.file.FileWritingMessageHandler;
import org.springframework.integration.file.support.FileExistsMode;
import org.springframework.integration.transformer.GenericTransformer;

@Configuration
public class FileWriterIntegrationConfig {

  @Bean
  @Transformer(inputChannel="textInChannel",       ←--- 声明转换器
               outputChannel="fileWriterChannel")
  public GenericTransformer<String, String> upperCaseTransformer() {
    return text -> text.toUpperCase();
  }

  @Bean
  @ServiceActivator(inputChannel="fileWriterChannel")
  public FileWritingMessageHandler fileWriter() {        ←--- 声明文件写
入器
    FileWritingMessageHandler handler =
        new FileWritingMessageHandler(new File("/tmp/sia5/files"));
    handler.setExpectReply(false);
    handler.setFileExistsMode(FileExistsMode.APPEND);
    handler.setAppendNewLine(true);
    return handler;
  }
}
```

在Java配置中，我们声明了两个bean：一个转换器，还有一个文件写入的消息处理器。这里的转换器是GenericTransformer。因为GenericTransformer是一个函数式接口，所以我们可以使用lambda表达式为其提供实现，这里调用了消息文本的toUpperCase()方法。转换器bean

使用了@Transformer注解，这样会将其声明成集成流中的一个转换器，它接受来自textInChannel通道的消息，然后将消息写入到名为fileWriterChannel的通道中。

负责文件写入的bean则使用了@ServiceActivator注解，表明它会接受来自fileWriter Channel的消息，并且会将消息传递给FileWritingMessageHandler实例所定义的服务。FileWritingMessageHandler是一个消息处理器，它会将消息的载荷写入特定目录的一个文件中，而文件的名称是通过消息的file_name头信息指定的。与XML样例类似，FileWritingMessageHandler也配置为以新行的方式为文件追加内容。

FileWritingMessageHandler bean的一个独特之处在于它调用了setExpectReply(false)方法，通过这个方法能够告知服务激活器（service activator）不要期望会有答复通道（reply channel，通过这样的通道，我们可以将某个值返回到流中的上游组件）。如果我们不调用setExpectReply()，文件写入bean的默认值是true。尽管管道的功能和预期一样，但是在日志中会看到一些错误日志，提示我们没有设置答复通道。

我们在这里没有必要显式声明通道。如果名为textInChannel和fileWriterChannel的bean不存在，那么这两个通道将会自动创建。但是，如果你想要更加精确地控制通道配置，那么可以按照如下的方式显式构建这些bean：

```
@Bean
```

```
public MessageChannel textInChannel() {
  return new DirectChannel();
}

…
@Bean
public MessageChannel fileWriterChannel() {
  return new DirectChannel();
}
```

基于Java的配置方案可能会更易于阅读，也更加简洁，而且符合我在本书中倡导的纯Java配置风格。但是，使用Spring Integration的Java DSL（Domain-Specific Language，领域特定语言）配置风格的话，它可以更加流畅。

9.1.3　使用Spring Integration的DSL配置

我们再次尝试一下文件写入集成流的定义。这一次，我们依然使用Java进行定义，但是会使用Spring Integration的Java DSL。这一次我们不再将流中的每个组件都声明为单独的bean，而是使用一个bean来定义整个流，如程序清单9.4所示。

程序清单9.4　为集成流的设计提供一个流畅的API

```
package sia5;
import java.io.File;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.integration.file.dsl.Files;
import org.springframework.integration.file.support.FileExistsMode;

@Configuration
```

```
public class FileWriterIntegrationConfig {

  @Bean
  public IntegrationFlow fileWriterFlow() {
    return IntegrationFlows
        .from(MessageChannels.direct("textInChannel"))     ←--- 入站通道
        .<String, String>transform(t -> t.toUpperCase())    ←--- 声明转换
器
        .handle(Files                                       ←--- 处理文件写
入
            .outboundAdapter(new File("/tmp/sia5/files"))
            .fileExistsMode(FileExistsMode.APPEND)
            .appendNewLine(true))
        .get();
  }
}
```

这种配置方式在一个bean方法中定义了整个流，已经做到了尽可能简洁。Integration Flows类初始化构建器API，我们可以通过这个API来定义流。

在程序清单9.4中，我们首先从名为textInchannel的通道接收消息，然后进入一个转换器，将消息载荷转换成大写形式。通过转换器之后，消息会由出站通道适配器来进行处理。这个适配器是由Spring Integration file模块的Files类型创建的。最后，通过对get()的调用返回要构建的IntegrationFlow。简而言之，这个bean方法定义了与XML和Java配置样例相同的集成流。

你可能已经发现，与Java配置样例类似，我们不需要显式声明通道bean。我们引用了textInChannel，如果该名字对应的通道不存在，那么Spring Integration会自动创建它。不过，我们也可以显式声明bean。

对于连接转换器和出站通道适配器的通道，我们甚至没有通过名字

引用它。如果需要显式配置通道，那么我们可以在流定义的时候通过调用channel()来引用它的名称：

```
@Bean
public IntegrationFlow fileWriterFlow() {
  return IntegrationFlows
      .from(MessageChannels.direct("textInChannel"))
      .<String, String>transform(t -> t.toUpperCase())
      .channel(MessageChannels.direct("fileWriterChannel"))
      .handle(Files
          .outboundAdapter(new File("/tmp/sia5/files"))
          .fileExistsMode(FileExistsMode.APPEND)
          .appendNewLine(true))
      .get();
}
```

在使用Spring Integration的Java DSL（与其他的fluent API类似）的时候，我们必须要巧妙地使用空格来保持可读性。在这里的样例中，我小心翼翼地使用缩进来保证代码块的可读性。对于更长、更复杂的流，我们甚至可以考虑将流的一部分抽取到单独的方法或子流中，以实现更好的可读性。

现在，我们已经看到了如何使用3种不同的方式来定义一个简单的流。接下来，我们回过头来看一下Spring Integration的全景。

9.2 Spring Integration功能概览

Spring Integration涵盖了大量的集成场景。如果想将所有的内容放到一章中，就像把一头大象装到信封里一样不现实。在这里，我只会向你展示Spring Integration这头大象的照片，而不是对Spring Integration进行面面俱到的讲解，目的就是让你能够了解它是如何运行的。随后，我

们将会再创建一个集成流，为Taco Cloud应用添加新的功能。

集成流是由一个或多个如下介绍的组件组成的。在继续编写代码之前，我们先看一下这些组件在集成流中所扮演的角色。

- 通道（channel）：将消息从一个元素传递到另一个元素。
- 过滤器（filter）：基于某些断言，条件化地允许某些消息通过流。
- 转换器（transformer）：改变消息的值和/或将消息载荷从一种类型转换成另一种类型。
- 路由器（router）：将消息路由至一个或多个通道，通常会基于消息的头信息进行路由。
- 切分器（splitter）：将传入的消息切割成两个或更多的消息，然后将每个消息发送至不同的通道；
- 聚合器（aggregator）：切分器的反向操作，将来自不同通道的多个消息合并成一个消息。
- 服务激活器（service activator）：将消息传递给某个Java方法来进行处理，并将返回值发布到输出通道上。
- 通道适配器（channel adapter）：将通道连接到某些外部系统或传输方式，可以接受输入，也可以写出到外部系统。
- 网关（gateway）：通过接口将数据传递到集成流中。

在定义文件写入集成流的时候，我们已经看到过其中的一些组件了。FileWriterGateway是一个网关，通过它，应用可以提交要写入文件的文本。我们还定义了一个转换器，将给定的文本转换成大写的形式，随后，我们定义一个出站通道适配器，它执行将文本写入文件的任务。这个流有两个通道，textInChannel和fileWriterChannel，它们将应用中的其他组件连接在了一起。现在，我们按照承诺快速看一下这些集成流组

件。

9.2.1　消息通道

消息通道是消息穿行集成通道的一种方式（参见图9.2）。它们是连接Spring Integration其他组成部分的管道。

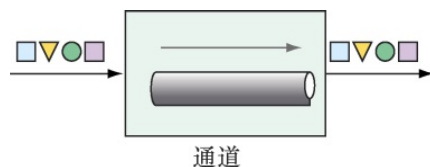


图9.2　消息通道是集成流中数据在其他组件之间流动的管道

Spring Integration提供了多种通道实现。

- PublishSubscribeChannel：发送到PublishSubscribeChannel的消息会被传递到一个或多个消费者中。如果有多个消费者，它们都会接收到消息。
- QueueChannel：发送到QueueChannel的消息会存储到一个队列中，会按照先进先出（First In First Out，FIFO）的方式被拉取出来。如果有多个消费者，只有其中的一个消费者会接收到消息。
- PriorityChannel：与QueueChannel类似，但它不是FIFO的方式，而是会基于消息的priority头信息被消费者拉取出来。
- RendezvousChannel：与QueueChannel类似，但是发送者会一直阻塞通道，直到消费者接收到消息为止，实际上会同步发送者和消费者。
- DirectChannel：与PublishSubscribeChannel类似，但是消息只会发送至一个消费者，它会在与发送者相同的线程中调用消费者。这种方式允许事务跨通道。

- ExecutorChannel：类似于DirectChannel，但是消息分发是通过TaskExecutor实现的，这样会在与发送者独立的线程中执行。这种通道类型不支持事务跨通道。
- FluxMessageChannel：反应式流的发布者消息通道，基于Reactor项目的Flux。（我们将会在第10章讨论反应式流、Reactor和Flux。）

在Java配置和Java DSL中，输入通道都是自动创建的，默认使用的是DirectChannel。但是，如果想要使用不同的通道实现，就需要将通道声明为bean并在集成流中引用它。例如，要声明PublishSubscribeChannel，我们需要声明如下的@Bean方法：

```
@Bean
public MessageChannel orderChannel() {
  return new PublishSubscribeChannel();
}
```

随后，我们可以在集成流定义中根据通道名称引用它。例如，这个通道要被一个服务激活器bean所消费，那么我们可以在@ServiceActivator注解的inputChannel属性中引用它：

```
@ServiceActivator(inputChannel="orderChannel")
```

或者，使用Java DSL配置风格，我们可以通过调用channel()来引用它：

```
@Bean
public IntegrationFlow orderFlow() {
  return IntegrationFlows
      ...
      .channel("orderChannel")
      ...
      .get();
}
```

很重要的一点需要注意，如果使用QueueChannel，消费者必须配置一个poller。例如，声明一个QueueChannel bean：

```
@Bean
public MessageChannel orderChannel() {
  return new QueueChannel();
}
```

那么，我们需要确保消费者配置成轮询该通道的消息。如果是服务激活器，@ServiceActivator注解可能会如下所示：

```
@ServiceActivator(inputChannel="orderChannel",
                  poller=@Poller(fixedRate="1000"))
```

在本例中，服务激活器每秒（或者说每1000毫秒）都会轮询名为orderChannel的通道。

9.2.2　过滤器

过滤器放置于集成管道的中间，它能够根据断言允许或拒绝消息进入流程的下一步（见图9.3）。

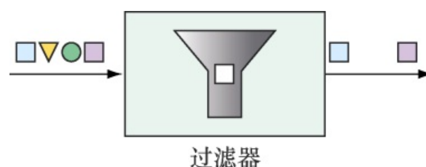


图9.3　过滤器会基于某个断言允许或拒绝消息在管道中进行处理

例如，假设消息包含了整型的值，它们要通过名为numberChannel的通道进行发布，但是我们只想让偶数进入名为evenNumberChannel的

通道。在这种情况下，我们可以使用@Filter注解定义一个过滤器：

```
@Filter(inputChannel="numberChannel",
        outputChannel="evenNumberChannel")
public boolean evenNumberFilter(Integer number) {
  return number % 2 == 0;
}
```

作为替代方案，如果使用Java DSL配置风格来定义集成流，那么我们可以按照如下的方式来调用filter()：

```
@Bean
public IntegrationFlow evenNumberFlow(AtomicInteger integerSource) {
  return IntegrationFlows
      ...
      .<Integer>filter((p) -> p % 2 == 0)
      ...
      .get();
}
```

在本例中，我们使用lambda表达式来实现过滤器。实际上，filter()方法接受GenericSelector作为参数。这意味着，如果我们的过滤器过于复杂，不适合放到一个简单的lambda表达式中，那么我们可以实现GenericSelector接口作为替代方案。

9.2.3 转换器

转换器会对消息执行一些操作，一般会形成不同的消息，有可能还会产生不同的载荷类型（见图9.4）。转换过程可能非常简单，比如执行数字的数学运算或者操作String值。转换也可能会很复杂，比如根据代表ISBN的String值查询并返回对应图书的详细信息。

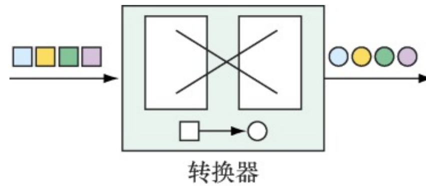


图9.4　转换器会改变流经集成流的消息

例如，假设整型值会通过名为numberChannel的通道进行发布，我们希望将这些数字转换成它们的罗马数字形式，以String类型来表示。在这种情况下，我们可以声明一个GenericTransformer类型的bean并为其添加@Transformer注解，如下所示：

```
@Bean
@Transformer(inputChannel="numberChannel",
             outputChannel="romanNumberChannel")
public GenericTransformer<Integer, String> romanNumTransformer() {
  return RomanNumbers::toRoman;
}
```

@Transformer注解可以将这个bean声明为转换器bean，它会从名为numberChannel的通道接收Integer值，然后使用静态方法toRoman()进行转换。（toRoman()是静态方法，定义在名为RomanNumbers的类中，这里通过方法引用来使用它。）转换后的结果会发布到名为romanNumberChannel的通道中。

在Java DSL配置风格中，调用transform()会更加简单，我们只需将对toRoman()的方法引用传递进来就可以了：

```
@Bean
public IntegrationFlow transformerFlow() {
  return IntegrationFlows
      ...
      .transform(RomanNumbers::toRoman)
```

```
        ...
        .get();
}
```

尽管在这两个转换器代码中我们都使用了方法引用，但是转换器也可以使用lambda表达式来进行声明。或者，如果转换器足够复杂，需要使用一个单独的类，那么我们可以将其作为一个bean注入流定义中，并将引用传递给transform()方法：

```
@Bean
public RomanNumberTransformer romanNumberTransformer() {
  return new RomanNumberTransformer();
}

@Bean
public IntegrationFlow transformerFlow(
                    RomanNumberTransformer romanNumberTransformer) {
  return IntegrationFlows
        ...
        .transform(romanNumberTransformer)
        ...
        .get();
}
```

在这里，我们声明了RomanNumberTransformer类型的bean，它本身是Spring Integration Transformer或GenericTransformer接口的实现。这个bean注入到了transformerFlow()方法中，并且在定义集成流的时候传递给了transform()方法。

9.2.4 路由器

路由器能够基于某个路由断言，实现集成流的分支，从而将消息发送至不同的通道上（见图9.5）。

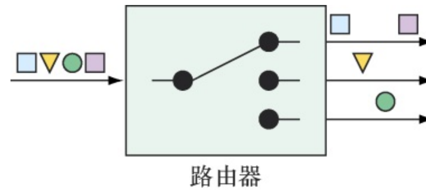


图9.5　路由器会根据应用于消息的断言将消息定向至不同的通道

例如，假设我们有一个名为numberChannel的通道，它会传输整型值。我们想要将带有偶数的消息定向到名为evenChannel的通道，将带有奇数的消息定向到名为oddChannel的通道。要在集成流中创建这样一个路由器，我们可以声明一个AbstractMessageRouter类型的bean，并为其添加@Router注解：

```
@Bean
@Router(inputChannel="numberChannel")
public AbstractMessageRouter evenOddRouter() {
  return new AbstractMessageRouter() {
    @Override
    protected Collection<MessageChannel>
              determineTargetChannels(Message<?> message) {
      Integer number = (Integer) message.getPayload();
      if (number % 2 == 0) {
        return Collections.singleton(evenChannel());
      }
      return Collections.singleton(oddChannel());
    }
  };
}

@Bean
public MessageChannel evenChannel() {
  return new DirectChannel();
}

@Bean
public MessageChannel oddChannel() {
  return new DirectChannel();
}
```

这里定义的AbstractMessageRouter接收名为numberChannel的输入通道的消息，以匿名内部类的形式检查消息的载荷：如果是偶数，就返回名为evenChannel的通道（在路由器bean之后同样以bean的方式进行了声明）；否则，通道载荷中的数字必然是奇数，将会返回名为oddChannel的通道（同样以bean方法的方式进行了声明）。

在Java DSL风格中，路由器是通过在流定义中调用route()方法来声明的，如下所示：

```
@Bean
public IntegrationFlow numberRoutingFlow(AtomicInteger source) {
  return IntegrationFlows
    ...
      .<Integer, String>route(n -> n%2==0 ? "EVEN":"ODD", mapping -> mappi
ng
        .subFlowMapping("EVEN", sf -> sf
            .<Integer, Integer>transform(n -> n * 10)
            .handle((i,h) -> { ... })
            )
        .subFlowMapping("ODD", sf -> sf
            .transform(RomanNumbers::toRoman)
            .handle((i,h) -> { ... })
            )
        )
      .get();
}
```

尽管我们依然可以定义AbstractMessageRouter并将其传递到route()，但是在这个样例中使用了lambda来确定消息载荷是偶数还是奇数。如果是偶数，就会返回值为EVEN的字符串；如果是奇数，就会返回值为ODD的字符串。然后这些值会用来确定该使用哪个子映射处理消息。

9.2.5　切分器

在集成流中，有时候将一个消息切分为多个消息独立处理可能会非常有用。切分器将会负责切分并处理这些消息，如图9.6所示。

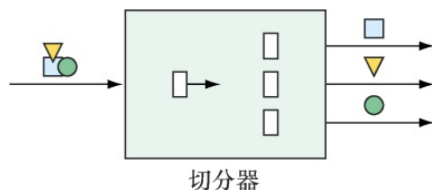


图9.6　切分器会将消息拆分为两个或更多独立的消息，它们可以由独立的子流分别进行处理

在很多场景中，切分器都非常有用，但是有两种基本的使用场景我们可以使用切分器。

- 消息载荷中包含了相同类型条目的一个列表，我们希望将它们作为单独的消息载荷来进行处理。例如，消息中携带了一个商品列表，它们可以切分为多个消息，每个消息的载荷分别对应一件商品。
- 消息载荷所携带的信息尽管有所关联，但是可以拆分为两个或更多不同类型的消息。例如，一个购买订单可能会包含投递信息、账单以及商品项的信息。投递细节可以通过某个子流来处理，账单由另一个子流来处理，而商品项由其他的子流来处理。在这种情况下，切分器后面通常会紧跟着一个路由器，它根据消息的载荷类型进行路由，确保数据都由正确的子流来进行处理。

在我们将消息载荷切分为两个或更多不同类型的消息时，通常定义一个POJO就足够了，它提取传入消息不同的组成部分，并以元素集合的形式返回。

例如，假设我们想要将带有购买订单的消息切分为两个消息：其中一个携带账单信息，另一个携带商品项的信息。如下的OrderSplitter就可以完成该任务：

```
public class OrderSplitter {
  public Collection<Object> splitOrderIntoParts(PurchaseOrder po) {
    ArrayList<Object> parts = new ArrayList<>();
    parts.add(po.getBillingInfo());
    parts.add(po.getLineItems());
    return parts;
  }
}
```

接下来，我们声明一个OrderSplitter bean，并通过@Splitter注解将其作为集成流的一部分：

```
@Bean
@Splitter(inputChannel="poChannel",
          outputChannel="splitOrderChannel")
public OrderSplitter orderSplitter() {
  return new OrderSplitter();
}
```

在这里，购买订单会到达名为poChannel的通道，它们会被OrderSplitter切分。然后，所返回集合中的每个条目都会作为集成流中独立的消息，它们会发布到名为splitOrderChannel的通道上。此时，我们可以在流中声明一个PayloadTypeRouter，将账单信息和商品项分别路由至它们自己的子流上：

```
@Bean
@Router(inputChannel="splitOrderChannel")
public MessageRouter splitOrderRouter() {
  PayloadTypeRouter router = new PayloadTypeRouter();
  router.setChannelMapping(
      BillingInfo.class.getName(), "billingInfoChannel");
```

```
  router.setChannelMapping(
      List.class.getName(), "lineItemsChannel");
  return router;
}
```

顾名思义，PayloadTypeRouter会根据消息的载荷将它们路由至不同的通道。按照这里的配置，载荷为BillingInfo类型的消息将会被路由至名为billingInfoChannel的通道，供后续进行处理。对于商品项来说，它们会放到一个java.util.List集合中，因此，我们将List类型的载荷映射到名为lineItemsChannel的通道中。

按照目前的状况，流将会被切分成两个子流：一个BillingInfo对象的流，另外一个则是List<LineItem>的流。如果我们想要进一步进行拆分，比如不想处理LineItems的列表，而是想要分别处理每个LineItem，又该怎么办呢？要将商品列表拆分为多个消息，其中每个消息包含一个条目，我们只需要编写一个方法（而不是一个bean）即可。这个方法带有@Splitter注解并且要返回LineItem的集合，如下所示：

```
@Splitter(inputChannel="lineItemsChannel", outputChannel="lineItemChannel"
)
public List<LineItem> lineItemSplitter(List<LineItem> lineItems) {
  return lineItems;
}
```

当带有List<LineItem>载荷的消息抵达名为lineItemsChannel的通道时，消息会进入lineItemSplitter()。按照切分器的规则，这个方法必须要返回切分后条目的集合。在本例中，我们已经有了LineItem的集合，所以我们直接返回这个集合就可以了。这样做的结果就是，集合中的每个LineItem都将会发布到一个消息中，这些消息会被发送到名为

lineItemChannel的通道中。

如果想要使用Java DSL声明相同的splitter/router配置，那么我们可以通过调用split()和route()来实现：

```
return IntegrationFlows
  ...
    .split(orderSplitter())
    .<Object, String> route(
        p -> {
          if (p.getClass().isAssignableFrom(BillingInfo.class)) {
            return "BILLING_INFO";
          } else {
            return "LINE_ITEMS";
          }
        }, mapping -> mapping
          .subFlowMapping("BILLING_INFO", sf -> sf
              .<BillingInfo> handle((billingInfo, h) -> {
                ...
              }))
          .subFlowMapping("LINE_ITEMS", sf -> sf
              .split()
              .<LineItem> handle((lineItem, h) -> {
                ...
              }))

        )
    .get();
```

DSL所组成的流定义相当简洁，但是可能会有点难以理解。它使用与Java配置样例相同的OrderSplitter来切分订单。在订单切分之后，它根据类型将其路由至两个独立的子流。

9.2.6 服务激活器

服务激活器接收来自输入通道的消息并将这些消息发送至一个

MessageHandler的实现，如图9.7所示。

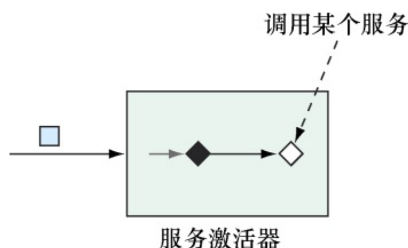


图9.7　在接收到消息时，服务激活器会通过MessageHandler调用某个服务

Spring Integration提供了多个开箱即用的MessageHandler（PayloadTypeRouter甚至就是MessageHandler的一个实现），但是我们通常会需要提供一些自定义的实现作为服务激活器。作为样例，如下的代码展现了如何声明MessageHandler bean并将其配置为服务激活器：

```
@Bean
@ServiceActivator(inputChannel="someChannel")
public MessageHandler sysoutHandler() {
  return message -> {
    System.out.println("Message payload: " + message.getPayload());
  };
}
```

这个bean使用了@ServiceActivator注解，表明它会作为一个服务激活器处理来自someChannel通道的消息。对于MessageHandler本身来讲，它是通过一个lambda表达式实现的。这是一个简单的MessageHandler，当得到消息之后，它会将消息的载荷打印至标准输出流。

另外，我们还可以声明一个服务激活器，让它在返回新载荷之前处理输入消息中的数据。在这种情况下，bean应该是一个

GenericHandler，而不是MessageHandler：

```
@Bean
@ServiceActivator(inputChannel="orderChannel",
                  outputChannel="completeOrder")
public GenericHandler<Order> orderHandler(
                             OrderRepository orderRepo) {
  return (payload, headers) -> {
    return orderRepo.save(payload);
  };
}
```

在本例中，服务激活器是一个GenericHandler，会接收载荷为Order类型的消息。当订单抵达时，我们会通过一个repository将它保存起来，并返回保存之后的Order，这个Order随后被发送至名为completeChannel的输出通道。

你可能已经注意到了，GenericHandler不仅能够得到载荷，还能得到消息头（虽然我们这个样例根本没有用到这些头信息）。我们还可以在Java DSL配置风格中使用服务激活器，此时，只需要将MessageHandler或GenericHandler传递到流定义的handle()方法中即可：

```
public IntegrationFlow someFlow() {
  return IntegrationFlows
    ...
      .handle(msg -> {
        System.out.println("Message payload: " + msg.getPayload());
      })
      .get();
}
```

在本例中，MessageHandler会得到一个lambda表达式，但是我们也可以为其提供一个方法引用，甚至是实现了MessageHandler接口的类实例。如果我们为其提供lambda表达式或方法引用，就需要记住它们均接

受消息作为其参数。

类似的，如果服务激活器不想成为流的终点，那么handle()还可以接受GenericHandler。如果要将前面提到的订单保存服务激活器添加进来，我们可以按照如下的形式使用Java DSL配置流：

```
public IntegrationFlow orderFlow(OrderRepository orderRepo) {
  return IntegrationFlows
    ...
      .<Order>handle((payload, headers) -> {
          return orderRepo.save(payload);
      })
    ...
      .get();
}
```

在使用GenericHandler的时候，lambda表达式或方法引用会接受消息负载和头信息作为参数。如果你选择使用GenericHandler作为流的终点，就需要返回null；否则，出现错误，提示没有指定输出通道。

9.2.7 网关

通过网关，应用可以提交数据到集成流中，并且能够可选地接收流的结果作为响应。网关会声明为接口，借助Spring Integration的实现，应用可以调用它来发送消息到集成流中（见图9.8）。

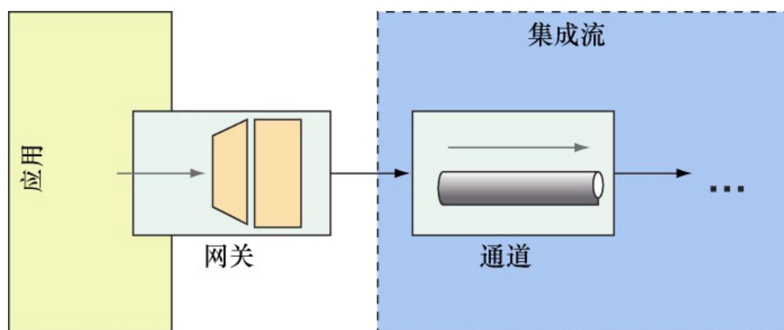


图9.8　服务网关是接口

我们已经见过了消息网关的样例，也就是FileWriterGateway。FileWriterGateway是一个单向的网关，它有一个接受String类型的方法，该方法会将文本写入到文件中，返回void。编写双向的网关同样简单。在编写网关接口的时候，只需确保方法要返回某个值，以便于推送到集成流中。

作为样例，假设有个网关，它面对的是一个简单的集成流，这个流会接受一个String并将给定的String转换成全大写的形式。这个网关接口大致如下所示：

```
package com.example.demo;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.stereotype.Component;
@Component
@MessagingGateway(defaultRequestChannel="inChannel",
                  defaultReplyChannel="outChannel")
public interface UpperCaseGateway {
  String uppercase(String in);
}
```

最让人开心的是，这个接口不需要实现。Spring Integration会自动在运行时提供一个实现，它会通过特定的通道发送和接收消息。

当uppercase()被调用的时候，给定的String会发布到集成流中，进入名为inChannel的通道。不管流是如何定义的或者它都干了些什么，当数据进入名为outChannel的通道时，它将会从uppercase()方法返回。

对于我们这个转换成大写格式的集成流来说，它是一个非常简单的流，只需要一个将String转换成大写格式的步骤就可以。它可以通过Java DSL配置声明如下：

```
@Bean
public IntegrationFlow uppercaseFlow() {
  return IntegrationFlows
    .from("inChannel")
    .<String, String> transform(s -> s.toUpperCase())
    .channel("outChannel")
    .get();
}
```

按照定义，这个流会从进入inChannel通道的数据开始。消息载荷会由转换器进行处理，也就是执行大写操作（通过lambda表达式来定义）。结果形成的消息会被发送到名为outChannel的通道，也就是我们在UpperCaseGateway中声明的答复通道。

9.2.8 通道适配器

通道适配器代表了集成流的入口和出口。数据通过入站通道适配器（inbound channel adapter）进入一个集成流，通过出站通道适配器离开一个集成流，如图9.9所示。

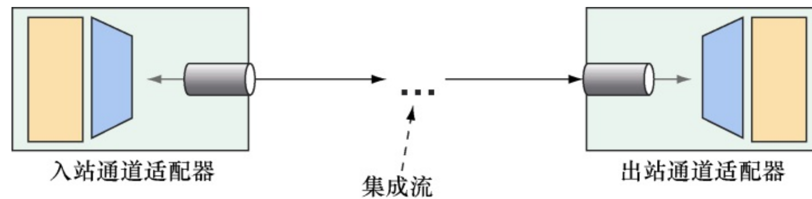


图9.9　通道适配器是集成流的入口和出口

根据要将什么数据源引入集成流，入站通道适配器可以有很多种形式。例如，我们可以声明一个入站通道适配器，将来自AtomicInteger 不断递增的数字引入到流中。如果使用Java配置，如下所示：

```
@Bean
@InboundChannelAdapter(
    poller=@Poller(fixedRate="1000"), channel="numberChannel")
public MessageSource<Integer> numberSource(AtomicInteger source) {
  return () -> {
    return new GenericMessage<>(source.getAndIncrement());
  };
}
```

这个@Bean方法通过@InboundChannelAdapter注解声明了一个入站通道适配器，根据注入的AtomicInteger每隔一秒（也就是1000毫秒）就提交一个数字给名为numberChannel的通道。

在使用Java配置时，我们可以通过@InboundChannelAdapter注解声明入站通道适配器，而在使用Java DSL定义集成流的时候，我们需要使用from()方法完成同样的事情。如下的流定义程序清单展现了类似的入站通道适配器，它是使用Java DSL定义的：

```
@Bean
public IntegrationFlow someFlow(AtomicInteger integerSource) {
  return IntegrationFlows
      .from(integerSource, "getAndIncrement",
          c -> c.poller(Pollers.fixedRate(1000)))
```

```
    ...
      .get();
}
```

通常，通道适配器是由Spring Integration的众多端点模块提供的。假设，我们需要一个入站通道适配器，它会监控一个特定的目录并将写入该目录的文件以消息的形式提交到file-channel通道中。如下的Java配置使用来自Spring Integration的file端点模块实现该功能：

```
@Bean
@InboundChannelAdapter(channel="file-channel",
                                           poller=@Poller(fixedDelay="1000
"))
public MessageSource<File> fileReadingMessageSource() {
  FileReadingMessageSource sourceReader = new FileReadingMessageSource();
  sourceReader.setDirectory(new File(INPUT_DIR));
  sourceReader.setFilter(new SimplePatternFileListFilter(FILE_PATTERN));
  return sourceReader;
}
```

如果使用Java DSL编写同等功能的入站通道适配器，那么我们可以使用Files类的inboundAdapter()。出站通道适配器是集成流的终点，会将最终的消息传递给应用或其他外部系统：

```
@Bean
public IntegrationFlow fileReaderFlow() {
  return IntegrationFlows
      .from(Files.inboundAdapter(new File(INPUT_DIR))
          .patternFilter(FILE_PATTERN))
      .get();
}
```

我们通常会将服务激活器实现为消息处理器，让它作为出站通道适配器，在数据需要传递给应用本身的时候更是如此。我们已经讨论过服务激活器，所以没有必要重复讨论了。

但是，需要注意，Spring Integration端点模块为多个通用场景提供了消息处理器。在程序清单 9.3 中，我们已经见到过一个这种出站通道适配器的样例，即FileWriting MessageHandler。提到Spring Integration端点模块，我们看一下都有哪些直接可用的集成端点模块。

9.2.9　端点模块

Spring Integration允许我们创建自己的通道适配器，这一点非常好，但是更棒的是Spring Integration提供了二十多个包含通道适配器（同时包括入站和出站的适配器）的端点模块（见表9.1），用于和各种常见的外部系统实现集成。

表**9.1**　**Spring Integration**提供的二十多个端点模块

模块	依赖的**artifact ID**（**Group ID**：**org.springframework.integration**）
AMQP	spring-integration-amqp
Spring应用事件	spring-integration-event
RSS和Atom	spring-integration-feed
文件系统	spring-integration-file
FTP/FTPS	spring-integration-ftp

GemFire	spring-integration-gemfire
HTTP	spring-integration-http
JDBC	spring-integration-jdbc
JPA	spring-integration-jpa
JMS	spring-integration-jms
Email	spring-integration-mail
MongoDB	spring-integration-mongodb
MQTT	spring-integration-mqtt
Redis	spring-integration-redis
RMI	spring-integration-rmi
SFTP	spring-integration-sftp
STOMP	spring-integration-stomp
Stream	spring-integration-stream

Syslog	spring-integration-syslog
TCP/UDP	spring-integration-ip
Twitter	spring-integration-twitter
Web Services	spring-integration-ws
WebFlux	spring-integration-webflux
WebSocket	spring-integration-websocket
XMPP	spring-integration-xmpp
ZooKeeper	spring-integration-zookeeper

从表9.1我们可以清楚地看到，Spring Integration提供了用途广泛的一组组件，它们能够满足非常多的集成需求。大多数应用程序所使用的只是Spring Integration所提供功能的九牛一毛。需要的话，我们最好还是要知道Spring Integration已经提供了相关的功能。

另外，我们不可能在一章的篇幅中介绍表9.1中的所有通道适配器。我们已经看到了如何使用文件系统模块写入文件的样例，随后将会看到如何使用Email模块来读取Email。

对于每个端点模块的通道适配器，我们可以在Java配置中将其声明为bean，也可以在Java DSL配置中通过静态方法的方式引用。我建议你探索一下自己感兴趣的其他端点模块。你会发现它们在使用方式上是非常一致的。现在，我们关注一下Email端点模块，看一下如何将它用到Taco Cloud应用中。

9.3　创建Email集成流

我们决定Taco Cloud应该允许客户通过Email提交taco设计和创建订单。我们发放传单并在报纸上刊登外卖广告，邀请每个人通过Email发送taco订单。这非常成功！但是，令人遗憾的是，它过于成功了。有太多的Email涌了进来，我们不得不申请临时帮助，让别人阅读所有的Email并将订单提交到订单系统中。

在本节中，我们将会实现一个集成流，轮询Taco Cloud的taco订单Email的收件箱、解析Email中的订单细节并将订单提交给Taco Cloud来进行处理。简而言之，在我们所创建的集成流中，入站通道适配器将会使用Email端点模块摄取Taco Cloud收件箱中的Email到集成流中。

集成流的下一步会将Email解析为订单对象，这些订单对象会被传递给另一个处理器，从而将订单提交至Taco Cloud的REST API中，在这里我们会像其他订单那样处理它们。首先，我们定义一个简单的配置属性类，它会捕获处理Taco Cloud Email的特定信息：

```
@Data
@ConfigurationProperties(prefix="tacocloud.email")
```

```
@Component
public class EmailProperties {

  private String username;
  private String password;
  private String host;
  private String mailbox;
  private long pollRate = 30000;

  public String getImapUrl() {
    return String.format("imaps://%s:%s@%s/%s",
        this.username, this.password, this.host, this.mailbox);
  }

}
```

我们可以看到，EmailProperties会捕获生成IMAP URL的属性。这个流会使用这个URL连接Taco Cloud Email服务器并轮询Email。在捕获的属性中包括Email用户的用户名和密码以及IMAP服务器的主机、要轮询的邮箱以及邮箱轮询的频率（默认为30秒）。

EmailProperties在类级别使用了@ConfigurationProperties注解，并将prefix属性设置为tacocloud.email。这意味着，我们可以在application.yml文件中按照下述方式配置消费Email的详细信息：

```
tacocloud:
  email:
    host: imap.tacocloud.com
    mailbox: INBOX
    username: taco-in-flow
    password: 1L0v3T4c0s
    poll-rate: 10000
```

现在，我们使用EmailProperties来配置集成流。我们想要创建的流大致如图9.10所示。

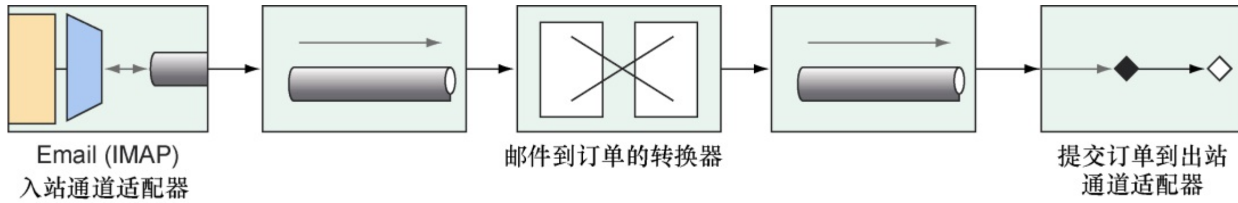


图9.10　通过Email接受taco订单的集成流

我们有两种方案来定义这个流。

- 在Taco Cloud应用中进行定义：在流的结束点，服务激活器要调用我们之前定义的创建订单的repository。
- 在单独的应用中进行定义：在流的结束点，服务激活器要发送POST请求到Taco Cloud API以提交taco订单。

不管选择哪种方式，除了服务激活器的实现方式之外，对流的本身影响并不大。但是，因为我们需要一些表示taco、订单和配料的类型，它们与Taco Cloud主应用可能会略微有所差异，所以我们会在一个单独的应用中定义集成流，避免与已有的领域类型相混淆。

我们还可以选择使用XML配置、Java配置或者Java DSL来定义流。我更喜欢DSL的优雅，所以在这里将会使用这种方案。如果你想要一些额外的挑战，也可以选择其他配置风格编写流的定义。现在，我们看一下taco Email订单流的Java DSL配置，如程序清单9.5所示。

程序清单9.5　定义接收Email并将其提交为订单的集成流

```
package tacos.email;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.Pollers;
```

```
@Configuration
public class TacoOrderEmailIntegrationConfig {

  @Bean
  public IntegrationFlow tacoOrderEmailFlow(
      EmailProperties emailProps,
      EmailToOrderTransformer emailToOrderTransformer,
      OrderSubmitMessageHandler orderSubmitHandler) {

  return IntegrationFlows
      .from(Mail.imapInboundAdapter(emailProps.getImapUrl()),
          e -> e.poller(
              Pollers.fixedDelay(emailProps.getPollRate())))
      .transform(emailToOrderTransformer)
      .handle(orderSubmitHandler)
      .get();
  }

}
```

根据tacoOrderEmailFlow()方法的定义，taco Email订单流由3个不同的组件组成。

- MAP Email入站通道适配器：使用IMP URL创建，而URL是根据EmailProperties的getImapUrl()方法创建的，并且会根据EmailProperties中设置的pollRate属性进行轮询。传入的Email会传递给一个通道，然后连接到转换器。
- 将Email转换成订单对象的转换器：转换器是通过EmailToOrderTransformer实现的，它会注入tacoOrderEmailFlow()方法中。转换所形成的订单会通过另外一个通道传递给最后一个组件。
- 处理器（作为出站通道适配器）：处理器接受订单对象并将其提交至Taco Cloud的REST API。

我们只有将Email端点模块作为依赖项添加到项目构建文件中，才

能调用Mail.imap InboundAdapter()。Maven依赖如下所示：

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
</dependency>
```

EmailToOrderTransformer是Spring Integration Transformer接口的实现，扩展了AbstractMailMessageTransformer（如程序清单9.6所示）。

程序清单9.6　使用集成转换器将传入的Email转换为taco订单

```
@Component
public class EmailToOrderTransformer
     extends AbstractMailMessageTransformer<Order> {

  @Override
  protected AbstractIntegrationMessageBuilder<Order>
                doTransform(Message mailMessage) throws Exception {
    Order tacoOrder = processPayload(mailMessage);
    return MessageBuilder.withPayload(tacoOrder);
  }

  ...

}
```

AbstractMailMessageTransformer是一个很便利的基类，适用于载荷为Email的消息。它会抽取传入消息Email的信息，并将它放到一个Message对象中，传递给doTransform()方法。在doTransform()方法中，我们将Message对象传递给一个名为processPayload()的private方法，将Email解析为Order对象。这个Order对象尽管和主Taco Cloud应用中的Order对象有些相似，但是并不完全相同，这里更加简单一些：

```
package tacos.email;
import java.util.ArrayList;
```

```
import java.util.List;
import lombok.Data;

@Data
public class Order {
  private final String email;
  private List<Taco> tacos = new ArrayList<>();

  public void addTaco(Taco taco) {
    this.tacos.add(taco);
  }
}
```

这个Order类不包含客户完整的投递信息和账单信息，而是只携带了客户的Email地址（通过传入的Email获取的）。

将Email解析成订单是一项非常重要的任务。实际上，即便最简单的实现也需要几十行代码。这些代码对于进一步讨论Spring Integration和如何实现转换器并没有任何助益。所以，为了节省空间，我在这里省略了processPayload()方法的细节。

EmailToOrderTransformer做的最后一件事情就是返回一个MessageBuilder，让消息的载荷中包含Order对象。MessageBuilder所生成的消息会发送至集成流的最后一个组件：将订单提交至Taco Cloud API的消息处理器。OrderSubmitMessageHandler实现了Spring Integration的GenericHandler，它会处理带有Order载荷的消息，如程序清单9.7所示。

程序清单9.7　通过消息处理器将订单提交至Taco Cloud API

```
package tacos.email;
import java.util.Map;
import org.springframework.integration.handler.GenericHandler;
import org.springframework.stereotype.Component;
```

```
import org.springframework.web.client.RestTemplate;

@Component
public class OrderSubmitMessageHandler
                                  implements GenericHandler<Order> {
  private RestTemplate rest;
  private ApiProperties apiProps;

  public OrderSubmitMessageHandler(
          ApiProperties apiProps, RestTemplate rest) {
    this.apiProps = apiProps;
    this.rest = rest;
  }

  @Override
  public Object handle(Order order, Map<String, Object> headers) {
    rest.postForObject(apiProps.getUrl(), order, String.class);
    return null;
  }
}
```

为了满足GenericHandler接口的要求，OrderSubmitMessageHandler重写了handle()方法，这个方法接收传入的Order对象，并使用注入的RestTemplate利用POST请求将Order提交至ApiProperties对象指定的URL。最后，handle()方法返回null，表明这个处理器是流的终点。

这里使用ApiProperties避免在postForObject()时硬编码URL。它是一个配置属性类，如下所示：

```
@Data
@ConfigurationProperties(prefix="tacocloud.api")
@Component
public class ApiProperties {
  private String url;
}
```

在application.yml中，Taco Cloud API的URL可能会配置如下：

```
tacocloud:
```

```
  api:
    url: http://api.tacocloud.com
```

为了让这个应用能够使用RestTemplate，并自动注入OrderSubmitMessageHandler中，我们需要在项目的构建文件中添加Spring Boot web starter依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

这不仅会将RestTemplate添加到类路径中，还会触发Spring MVC的自动配置功能。作为独立的Spring Integration流，这个应用并不需要Spring MVC，更不需要自动配置所提供的嵌入式Tomcat。所以，我们可以在application.yml中通过如下的配置条目禁用Spring MVC的自动配置：

```
spring:
  main:
    web-application-type: none
```

spring.main.web-application-type属性可以设置为servlet、reactive或none。当Spring MVC位于类路径之中时，自动配置功能会将其设置为servlet。我们在这里将其重写为none，所以Spring MVC和Tomcat将不会进行自动配置（我们将会在第11章介绍反应式Web应用是什么样子的）。

9.4　小结

- 借助Spring Integration能够定义流，在进入和离开应用的时候可以对数据进行处理。
- 集成流可以使用XML、Java或简洁的Java DSL配置风格来进行定义。
- 消息网关和通道适配器会作为集成流的入口和出口。
- 在流动的过程中，消息可以进行转换、切分、聚合、路由，也可以由服务激活器对其进行处理。
- 消息通道连接集成流中的各个组件。

第3部分　反应式Spring

在第3部分，我们将探索 Spring 对反应式编程提供的全新支持。第10章讨论使用Reactor项目进行反应式编程的基础知识。Reactor项目是支撑 Spring 5 反应式特性的反应式编程库。然后，我们将会介绍Reactor 中一些常用的反应式操作。在第11章中，我们会重新探讨REST API的开发，介绍全新的Web框架Spring WebFlex。该框架借用了很多Spring MVC的理念，为Web开发提供了新的反应式模型。第12章总结第3部分，介绍如何通过Spring Data对Cassandra和Mongo数据库进行读写，实现反应式数据持久化。

第10章　理解反应式编程

本章内容：

- 反应式编程概览
- Reactor项目简介
- 反应式地处理数据

你曾有过订阅报纸或者杂志的经历吗？互联网的确从传统的出版发行商那儿分得了一杯羹，但是过去订阅报纸真的是我们了解时事的最佳方式。那时，我们每天早上都会收到一份新鲜出炉的报纸，并在早饭时间或上班路上阅读。

现在假设一下，在支付完订阅费用之后，几天的时间过去了，你却没有收到任何报纸。又过了几天，你打电话给报社的销售部门询问为什么还没有收到报纸。想象一下，如果他们告诉你："因为你支付的是一整年的订阅费用，而现在这一年还没有结束，当这一年结束时，你肯定可以一次性完整地收到它们。"那么你会有多么惊讶。

值得庆幸的是，这并非订阅的真正运作方式。报纸具有一定的时效性。在出版后，报纸需要及时投递，以确保在阅读它们时内容仍然是新鲜的。此外，当你在阅读最新一期的报纸时，记者们正在为未来的版本撰写内容，同时印刷机正在满速运转，印刷下一期的内容——一切都是并行的。

在开发应用程序代码时，我们可以编写两种风格的代码，即命令式和反应式。

- 命令式（Imperative）的代码：非常类似于上文所提的虚构的报纸订阅方式。它由一组任务组成，每次只运行一项任务，每项任务又都依赖于前面的任务。数据会按批次进行处理，在前一项任务还没有完成对当前数据批次的处理时，不能将这些数据递交给下一项处理任务。
- 反应式（Reactive）的代码：非常类似于真实的报纸订阅方式。它定义了一组用来处理数据的任务，但是这些任务可以并行地执行。每项任务处理数据的一部分子集，并将结果交给处理流程中的下一项任务，同时继续处理数据的另一部分子集。

在本章中，我们将暂别Taco Cloud应用程序，转而探索Reactor项目。Reactor 是一个反应式编程库，同时也是Spring家族的一部分。它是Spring 5反应式编程功能的基础，所以在我们学习使用Spring构建反应式控制器和repository之前，理解 Reactor是非常重要的。不过，在我们开始学习Reactor之前，还需要花点时间研究一下反应式编程的基本要素。

10.1 反应式编程概览

反应式编程是一种可以替代命令式编程的编程范式。这种可替代性存在的原因在于反应式编程解决了命令式编程中的一些限制。理解这些限制，有助于你更好地理解反应式编程模型的优点。

> 注意：反应式编程不是银弹。你不应该从这一章或者其他任何关于反应式编程的讨论中得出“命令式编程是邪恶的，而反应式编程才是你的救星”的结论。如同我们作为开发者学习到的任何技术一样，反应式编程对于某些使用场景来说的确是完美的，但是在其他的一些场景中可能不那么适用。建议以实用主义为上。

如果你和我以及绝大多数的开发者一样，是从命令式编程开始入行的，那么很可能你现在编写的大部分（或者所有）代码在将来依然是命令式的。命令式编程相当直观，没有编程经验的学生们可以在学校的STEM教育课程中轻松地学习它，而且足够强大。在驱动大型企业运行的代码中，绝大部分都是命令式的。

它的理念很简单：你可以一次一个地按照顺序将代码编写为需要遵循的指令列表。在某项任务开始执行之后，程序在开始下一项任务之前需要等待当前任务完成。在整个处理过程中的每一步，要处理的数据都必须是完全可用的，以便将它们作为一个整体进行处理。

一开始一切都很美好，直到我们遇到问题。在执行一项任务的时

候，特别是 IO 任务（将数据写入 DB 或者从远程服务器获取数据），触发这项任务的线程实际上是被阻塞的，在任务完成之前它不能做任何事情。坦白来说，阻塞线程是一种浪费。

大多数编程语言（包括Java）都支持并发编程。在Java中创建一个线程，然后让它执行某些操作，而调用线程继续执行其他工作，这是相当容易实现的。虽然创建线程很简单，但是这些线程多半最终会被阻塞。管理多线程中的并发极具挑战，而更多线程则意味着更多的复杂性。

相比之下，反应式编程本质上是函数式的和声明式的。相对于描述一组将依次执行的步骤，反应式编程描述了数据将会流经的管道或者流。相对于要求将被处理的数据作为一个整体进行处理，反应式流可以在数据可用时立即开始处理。实际上，传入的数据可能是无限的（比如，一个某个地理位置的实时温度测量数据的恒定流）。

拿现实世界类比一下，可以将命令式编程看作是水气球，而将反应式编程看作是花园里的软管。在夏天，这两者都是偷袭和愉悦毫无戒心的朋友的好方式，但是它们的运作方式却不同：

- 水气球只能一次性地填满有效载荷，并在撞到目标时弄湿对象。水气球的容量有限，如果你想要弄湿更多人（或者把同一个人弄得更加湿透一点），那么唯一的选择就是增加水气球的数量。
- 花园软管的有效载荷是从水龙头到喷嘴的水流。在特定的时间点，花园软管的容量可能是有限的，但是在打水仗的过程中它的容量却是无限的。只要水源源不断地从龙头流入软管中，水就会源源不断

地从喷嘴喷出去。同一个软管非常好扩展，你可以尽情地和更多的朋友打水仗。

虽然水气球（或者命令式编程）没有什么固有的问题，但是持有软管（或者能够应用反应式编程）的人通常在伸缩性和性能方面更具优势。

定义反应式流

反应式流（Reactive Streams）是由Netflix、Lightbend和Pivotal（Spring背后的公司）的工程师于2013年年底开始制定的一种规范。反应式流旨在提供无阻塞回压的异步流处理标准。

我们已经触及了反应式编程的异步特性，它使我们能够并行执行任务，从而实现更高的可伸缩性。通过回压，数据消费者可以限制它们想要处理的数据数量，避免被过快的数据源所淹没。

Java的流和反应式流

Java的流和反应式流之间有很多相似之处。首先，它们的名字中都有流（Stream）这个词。它们还提供了用于处理数据的函数式API。 事实上，正如你稍后将会在我们介绍Reactor时看到的那样，它们甚至可以共享许多相同的操作。

Java的流通常都是同步的，并且只能处理有限的数据集。从

本质上来说，它们只是使用函数来对集合进行迭代的一种方式。

反应式流支持异步处理任意大小的数据集，同样也包括无限数据集。只要数据就绪，它们就能实时地处理数据，并且能够通过回压来避免压垮数据的消费者。

反应式流规范可以总结为4个接口：Publisher、Subscriber、Subscription和Processor。Publisher负责生成数据，并将数据发送给Subscription（每个Subscriber对应一个Subscription）。Publisher接口声明了一个方法 subscribe()，Subscriber可以通过该方法向 Publisher发起订阅。

```
public interface Publisher<T> {
  void subscribe(Subscriber<? super T> subscriber);
}
```

一旦Subscriber订阅成功，就可以接收来自Publisher的事件。这些事件是通过Subscriber接口上的方法发送的：

```
public interface Subscriber<T> {
  void onSubscribe(Subscription sub);
  void onNext(T item);
  void onError(Throwable ex);
  void onComplete();
}
```

Subscriber的第一个事件是通过对 onSubscribe()方法的调用接收的。Publisher调用 onSubscribe() 方法时，会将Subscription对象传递给Subscriber。通过Subscription，Subscriber可以管理其订阅情况：

```
public interface Subscription {
  void request(long n);
  void cancel();
}
```

Subscriber可以通过调用 request()方法来请求 Publisher 发送数据，或者通过调用 cancel()方法表明它不再对数据感兴趣并且取消订阅。当调用 request()时，Subscriber 可以传入一个long类型的数值以表明它愿意接受多少数据。这也是回压能够发挥作用的地方，以避免Publisher 发送多于 Subscriber能够处理的数据量。在Publisher发送完所请求数量的数据项之后，Subscriber可以再次调用 request()方法来请求更多的数据。

Subscriber 请求数据之后，数据就会开始流经反应式流。Publisher 发布的每个数据项都会通过调用Subscriber 的 onNext()方法递交给 Subscriber。如果有任何错误，就会调用 onError()方法。如果 Publisher 没有更多的数据，也不会继续产生更多的数据，那么将会调用 Subscriber 的onComplete()方法来告知Subscriber 它已经结束。

至于Processor接口，它是Subscriber和Publisher的组合，如下所示：

```
public interface Processor<T, R>
        extends Subscriber<T>, Publisher<R> {}
```

当作为 Subscriber时，Processor会接收数据并以某种方式对数据进行处理。然后它会将角色转变为Publisher，并将处理的结果发布给它的Subscriber。

正如你所看到的，反应式流的规范非常简单，很容易就能想出如何构建一个以Publisher作为开始的数据处理管道，并让数据通过零个或多

个Processor，然后将最终结果投递给Subscriber。

然而，反应式流规范的接口本身并不支持以函数式的方式组成这样的流。Reactor项目是反应式流规范的一个实现，提供了一组用于组装反应式流的函数式API。我们将会在后面的内容中看到，Reactor构成了Spring 5反应式编程模型的基础。在本章的其余部分，我们将会探讨（并且，我敢说这个过程非常有意思）Reactor项目。

10.2 初识Reactor

反应式编程要求我们采取和命令式编程不一样的思维方式。此时我们不会再描述每一步要进行的步骤，反应式编程意味着要构建数据将要流经的管道。当数据流经管道时，可以对它们进行某种形式的修改或者使用。

例如，假设我们想要接受一个英文人名，然后将所有的字母都转换为大写，并用得到的结果创建一个问候消息，并最终打印它。使用命令式编程模型，代码看起来如下所示：

```
String name = "Craig";
String capitalName = name.toUpperCase();
String greeting = "Hello, " + capitalName + "!";
System.out.println(greeting);
```

使用命令式编程模型，每行代码执行一个步骤，按部就班，并且肯定在同一个线程中进行。每一步在执行完成之前都会阻止执行线程执行下一步。

与之不同，如下的函数式、反应式代码完成了相同的事情：

```
Mono.just("Craig")
    .map(n -> n.toUpperCase())
    .map(cn -> "Hello, " + cn + "!")
    .subscribe(System.out::println);
```

不用过度关心这个例子中的细节，我们很快将会详细讨论 just()、map()和subscribe() 方法。现在，重要的是要理解：虽然这个反应式的例子看起来依然保持着按步骤执行的模型，但实际是数据会流经处理管线。在处理管线的每一步，都对数据进行了某种形式的加工，但是我们不能判断数据会在哪个线程上执行操作。它们既可能在同一个线程，也可能在不同的线程。

这个例子中的Mono是Reactor的两种核心类型之一，另一个类型是Flux。 两者都实现了反应式流的Publisher接口。Flux代表具有零个、一个或者多个（可能是无限个）数据项的管道。Mono是一种特殊的反应式类型，针对数据项不超过一个的场景，它进行了优化。

Reactor与**RxJava**（**ReactiveX**）的对比

如果你熟悉RxJava或者ReactiveX，那么你可能认为Mono和Flux类似于Observable 和 Single。事实上它们不仅在语义上大致相同，还共享了很多相同的操作符。

虽然我们在本书中主要介绍Reactor，但是Reactor和RxJava的类型可以互相转换，我相信你对这一点会感到很开心。甚至，

在接下来的章节中我们还会看到，Spring 也可以使用RxJava的类型。

实际上，在前面的例子中有3个Mono。其中，just() 操作创建了第一个Mono。当该Mono发送一个值的时候，这个值被传递给了将字母转换为大写的map()操作，据此又创建了另一个Mono。当第二个Mono发布它的数据时，数据被传递给了第二个map()操作，并且会在此进行一些字符串连接操作，而结果将用于创建第三个Mono。最后，对第三个Mono上的subscribe()方法调用时，会接收数据并将数据打印出来。

10.2.1　绘制反应式流图

反应式流程通常使用弹珠图（marble diagram）表示，如图10.1所示。弹珠图的展现形式非常简单，在顶部描述了数据流经Flux或者Mono的时间线，在中间描述了要执行的操作，在底部描述了结果形成的Flux或者Mono的时间线。我们将会看到，当数据流经原始的Flux时，某些操作将会对它进行处理，并产生一个新的Flux，已经处理过的数据将会在新Flux中流动。

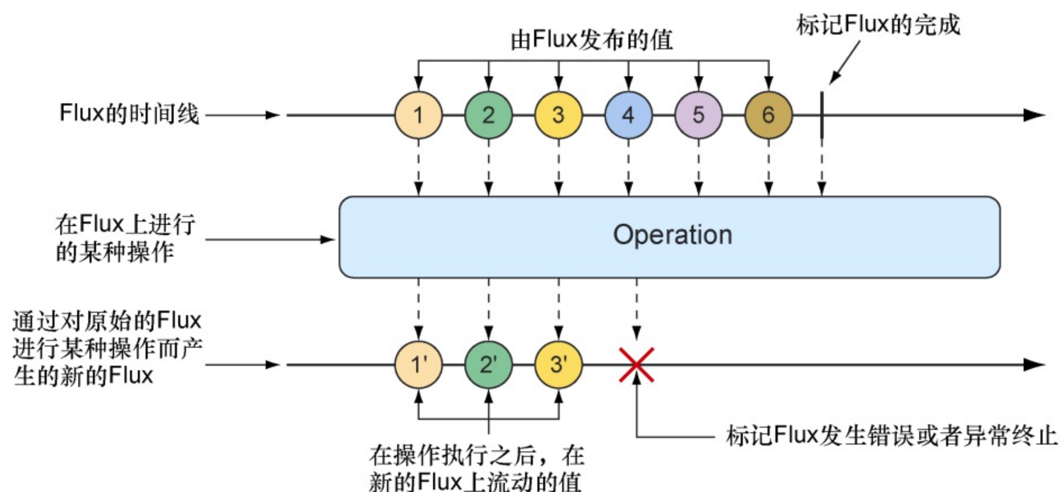


图10.1　描绘Flux基本流程的弹珠图

图10.2展示了一个类似的弹珠图，但是针对的是Mono。我们可以看到，这里主要的不同是Mono将会有零个或者一个数据项，或者一个错误。

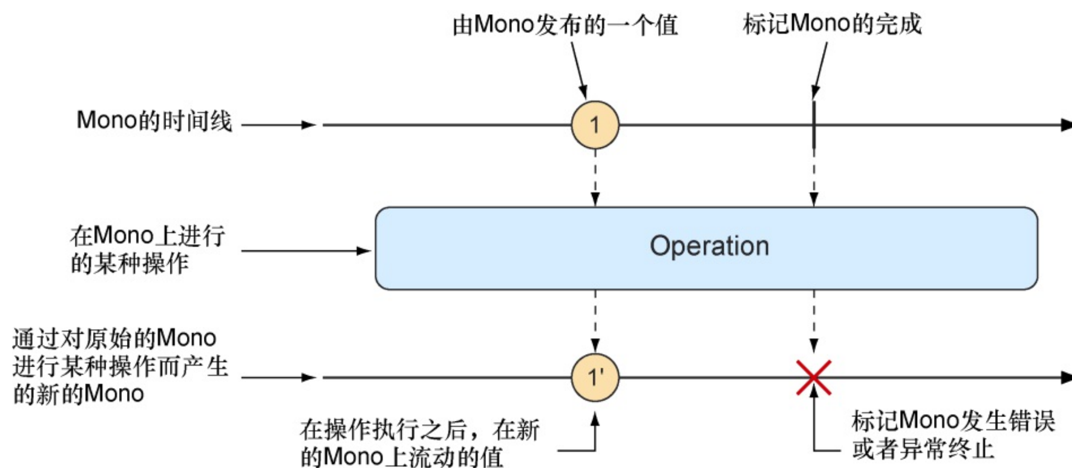


图10.2　描绘Mono基本流程的弹珠图

在10.3节，我们将会探索Flux和Mono支持的许多操作，同时我们还将使用弹珠图来可视化它们的工作原理。

10.2.2　添加Reactor依赖

要开始使用Reactor，请将下面的依赖项添加到项目的构建文件中：

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
</dependency>
```

Reactor还提供了非常棒的测试支持。我们将会围绕Reactor代码编写大量的测试，因此绝对需要将下面的依赖添加到构建文件中：

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-test</artifactId>
  <scope>test</scope>
</dependency>
```

如果你计划将这些依赖添加到一个Spring Boot工程中，那么Spring Boot工程会替你管理依赖。但是，如果要在非Spring Boot项目中使用Reactor，就需要在构建文件中设置Reactor的BOM（Bill Of Materials，物料清单）。下面的依赖管理条目将会把Reactor的Bismuth版本添加到构建文件中：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>Bismuth-RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

现在Reactor已经位于工程的构建文件中了，我们可以开始使用Mono和Flux来创建反应式的处理管线。在本章的剩余部分，我们将介绍Mono和Flux所提供的几个操作。

10.3　使用常见的反应式操作

Flux和Mono是Reactor提供的最基础的构建块，而这两种反应式类型所提供的操作符则是组合使用它们以构建数据流动管线的黏合剂。Flux和Mono共有500多个操作，这些操作都可以大致归类为：

- 创建操作；
- 组合操作；
- 转换操作；
- 逻辑操作。

虽然对这500多个操作一一探讨会非常有趣，但是本章的篇幅有限，所以我在本节中选择一些有用的操作来进行说明。下面让我们从创建操作开始吧。

> 注意：Mono的例子呢？因为Mono和Flux的很多操作是相同的，所以没有必要针对Mono和Flux重复进行介绍。此外，虽然Mono的操作也很有用，但是相比而言，Flux上的操作更有趣。我们的大多数示例会使用Flux。读者只需要知道Mono上通常具有相同的名称的操作即可。

10.3.1　创建反应式类型

在Spring中使用反应式类型时，我们通常将会从repository或者service中获取Flux或Mono，并不需要我们自行创建。偶尔，我们可能需要创建一个新的反应式Publisher。

Reactor提供了多种创建Flux和Mono的操作。这里，我们将介绍其中一些有用的创建操作。

根据对象创建

如果你有一个或多个对象，并想据此创建Flux或Mono，那么可以使用Flux或Mono上的静态 just()方法来创建一个反应式类型，它们的数据会由这些对象来驱动。例如，下面的测试方法将从5个String对象中创建一个Flux：

```
@Test
public void createAFlux_just() {
  Flux<String> fruitFlux = Flux
      .just("Apple", "Orange", "Grape", "Banana", "Strawberry");
}
```

此时我们已经创建了Flux，但是它还没有订阅者。如果没有任何的订阅者，那么数据将不会流动。回想一下花园软管的比喻，假设我们已经将花园软管连接到了水龙头上，另一侧是水厂的水——但是在你打开水龙头之前，水不会流动。订阅反应式类型就如同你打开数据流的水龙头。

要添加一个订阅者，我们可以在Flux上调用subscribe()方法：

```
fruitFlux.subscribe(
  f -> System.out.println("Here's some fruit: " + f)
);
```

这里传递给 subscribe()方法的lambda表达式实际上是一个java.util.Consumer，用来创建反应式流的Subscriber。在调用subscribe()之后，数据会开始流动。在这个例子中，没有中间操作，所以数据从Flux直接流向订阅者。

将来自Flux或Mono的数据项打印到控制台是观察反应式类型运行方式的好方法，实际测试Flux或Mono更好的方法是使用Reactor提供的StepVerifier。对于给定的Flux或Mono，StepVerifier将会订阅该反应式类型，在数据流过时对数据应用断言，并在最后验证反应式流是否按预期完成。

例如，要验证预定义的数据是否流经了fruitFlux，我们可以编写如下所示的测试代码：

```
StepVerifier.create(fruitFlux)
    .expectNext("Apple")
    .expectNext("Orange")
    .expectNext("Grape")
    .expectNext("Banana")
    .expectNext("Strawberry")
    .verifyComplete();
```

在这个例子中，StepVerifier订阅了fruitFlux，然后断言Flux中的每个数据项是否与预期的水果名称相匹配。最后，它验证Flux在发布完“Strawberry”之后，整个fruitFlux正常完成。

对于本章的其他例子，你可以使用StepVerifier来编写测试，验证Flux或者Mono行为，研究相应的工作原理，从而帮助你学习和了解Reactor中最有用的操作。

根据集合创建

我们还可以根据数组、Iterable 或者 Java Stream创建Flux。图10.3使用弹珠图展示如何使用这种方式进行创建。

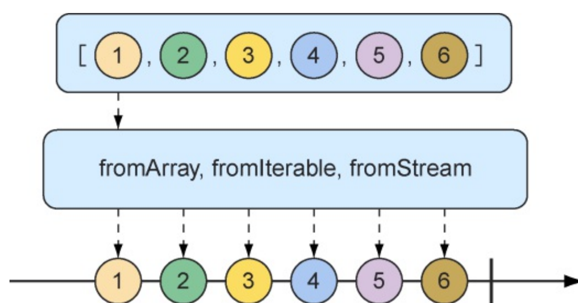


图10.3　可以根据数组、Iterable 或者Java Stream创建Flux

要根据数组创建Flux，可以调用Flux上的静态方法fromArray()，并传入一个源数组：

```
@Test
public void createAFlux_fromArray() {
  String[] fruits = new String[] {
      "Apple", "Orange", "Grape", "Banana", "Strawberry" };

  Flux<String> fruitFlux = Flux.fromArray(fruits);

  StepVerifier.create(fruitFlux)
      .expectNext("Apple")
      .expectNext("Orange")
      .expectNext("Grape")
      .expectNext("Banana")
      .expectNext("Strawberry")
      .verifyComplete();
```

```
}
```

因为该源数组包含了之前从对象列表创建Flux时所使用的相同的水果名称，所以该Flux发布的数据会有相同的值，可以使用和之前相同的StepVerifier来验证。

如果我们需要根据java.util.List、java.util.Set或者其他任意java.lang.Iterable 的实现来创建Flux，那么可以将其传递给静态的fromIterable()方法：

```
@Test
public void createAFlux_fromIterable() {
  List<String> fruitList = new ArrayList<>();
  fruitList.add("Apple");
  fruitList.add("Orange");
  fruitList.add("Grape");
  fruitList.add("Banana");
  fruitList.add("Strawberry");

  Flux<String> fruitFlux = Flux.fromIterable(fruitList);

  // ... verify steps
}
```

或者，我们有一个Java Stream，并且希望将其用作Flux的源，那么可以调用fromStream()方法：

```
@Test
public void createAFlux_fromStream() {
  Stream<String> fruitStream =
       Stream.of("Apple", "Orange", "Grape", "Banana", "Strawberry");

  Flux<String> fruitFlux = Flux.fromStream(fruitStream);

  // ... verify steps
}
```

同样，我们可以使用和之前一样的StepVerifier来验证该Flux发布的数据。

生成**Flux**的数据

有时候我们根本没有可用的数据，而只是想要一个作为计数器的Flux，它会在每次发送新值时增加1。要创建一个计数器Flux，我们可以使用静态方法range()。图10.4说明了range()方法的工作原理。

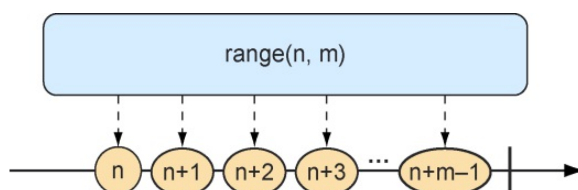


图10.4　从区间创建的Flux会以类似计数器的方式发布消息

下面的测试方法展示了如何创建一个区间Flux：

```
@Test
public void createAFlux_range() {
  Flux<Integer> intervalFlux =
      Flux.range(1, 5);
  StepVerifier.create(intervalFlux)
      .expectNext(1)
      .expectNext(2)
      .expectNext(3)
      .expectNext(4)
      .expectNext(5)
      .verifyComplete();
}
```

在这个例子中，我们创建了一个区间Flux，起始值为1，结束值为5。StepVerifier 证明了它将发布5个条目，即整数1到5。

另一个与range()方法类似的Flux创建方法是interval()。与range()方法一样，interval()方法会创建一个发布递增值的Flux。interval()的特殊之处在于，我们不是给它设置一个起始值和结束值，而是指定一个应该每隔多长时间发出值的间隔时间。图 10.5展示了interval()方法创建Flux的弹珠图。

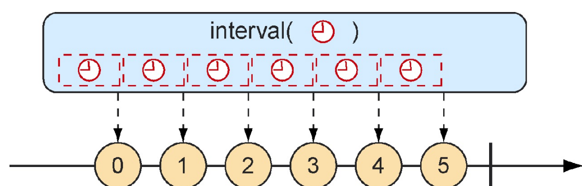


图10.5　根据指定间隔创建的Flux会周期性地发布条目

例如，要创建一个每秒发布一个值的Flux，你可以使用Flux上的静态interval() 方法，如下所示：

```
@Test
public void createAFlux_interval() {
  Flux<Long> intervalFlux =
      Flux.interval(Duration.ofSeconds(1))
          .take(5);

  StepVerifier.create(intervalFlux)
      .expectNext(0L)
      .expectNext(1L)
      .expectNext(2L)
      .expectNext(3L)
      .expectNext(4L)
      .verifyComplete();
}
```

需要注意的是，通过interval()方法创建的Flux会从0开始发布值，并且后续的条目依次递增。此外，因为interval()方法没有指定最大值，所以它可能会永远运行。我们也可以使用take()方法将结果限制为前5个条

目。我们将在下一节中详细讨论take()方法。

10.3.2　组合反应式类型

有时候，我们会需要操作两种反应式类型，并以某种方式将它们合并在一起。或者，在其他情况下，我们可能需要将Flux拆分为多种反应式类型。在本节中，我们将研究组合以及拆分Reactor的Flux和Mono的操作。

合并反应式类型

假设我们有两个Flux流，并且需要据此创建一个结果Flux，这个形成的Flux会在任意上游Flux流有数据时产生数据。要将一个Flux与另一个Flux合并，可以使用mergeWith()方法，如图10.6中的弹珠图所示。

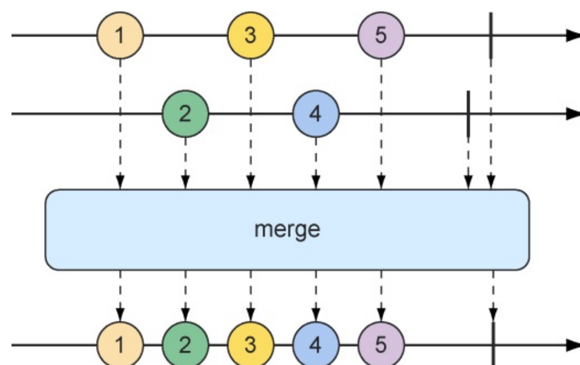


图10.6　合并两个Flux流（它们的消息将会交错合并为一个新的Flux）

例如，假设有一个值是电视和电影角色名称的Flux，还有另一个值是这些角色喜欢吃的食物的名称的Flux。下面的测试方法展示了如何使用mergeWith()方法合并两个Flux对象：

```
@Test
public void mergeFluxes() {

  Flux<String> characterFlux = Flux
      .just("Garfield", "Kojak", "Barbossa")
      .delayElements(Duration.ofMillis(500));
  Flux<String> foodFlux = Flux
      .just("Lasagna", "Lollipops", "Apples")
      .delaySubscription(Duration.ofMillis(250))
      .delayElements(Duration.ofMillis(500));

  Flux<String> mergedFlux = characterFlux.mergeWith(foodFlux);

  StepVerifier.create(mergedFlux)
      .expectNext("Garfield")
      .expectNext("Lasagna")
      .expectNext("Kojak")
      .expectNext("Lollipops")
      .expectNext("Barbossa")
      .expectNext("Apples")
      .verifyComplete();
}
```

通常，Flux 会尽可能快地发布数据。因此，我们在创建的两个Flux流上使用delayElements()方法来减慢它们的速度——每500毫秒发布一个条目。此外，为了使食物Flux在角色名称Flux之后再开始流式传输，我们调用了食物Flux上的delaySubscription()方法，以便它在订阅后再经过250毫秒后才开始发布数据。

在合并了两个Flux对象后，将会创建一个新的合并过后的Flux。当StepVerifier订阅这个合并过后的Flux时，它将依次订阅两个源Flux流并启动数据流。

这个合并过后的Flux数据项发布顺序与源Flux的发布时间一致。因为两个Flux对象都设置为以常规速率进行发布，所以这些值在合并后的Flux中会交错在一起，结果是：一个角色、一个食物、另一个角色、另

一个食物，以此类推。如果任何一个Flux的计时发生变化，那么你可能会看到接连发布了两个角色或者两个食物。

因为mergeWith()方法不能完美地保证源Flux之间的先后顺序，所以我们可以考虑使用zip()方法。当两个Flux对象压缩在一起的时候，它将会产生一个新的发布元组的Flux，其中每个元组中都包含了来自每个源Flux的数据项。图10.7说明了如何将两个Flux对象压缩在一起。

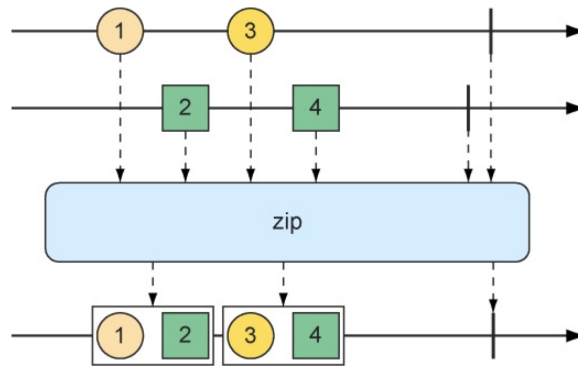


图10.7　通过zip()方法合并两个Flux流

要查看zip()操作实际如何运行，可以考虑如下所示的测试方法，它将角色Flux和食物Flux合并在一起：

```
@Test
public void zipFluxes() {
  Flux<String> characterFlux = Flux
      .just("Garfield", "Kojak", "Barbossa");
  Flux<String> foodFlux = Flux
      .just("Lasagna", "Lollipops", "Apples");

  Flux<Tuple2<String, String>> zippedFlux =
      Flux.zip(characterFlux, foodFlux);
  StepVerifier.create(zippedFlux)
      .expectNextMatches(p ->
          p.getT1().equals("Garfield") &&
          p.getT2().equals("Lasagna"))
      .expectNextMatches(p ->
```

```
            p.getT1().equals("Kojak") &&
            p.getT2().equals("Lollipops"))
        .expectNextMatches(p ->
            p.getT1().equals("Barbossa") &&
            p.getT2().equals("Apples"))
            .verifyComplete();
}
```

需要注意的是，与mergeWith()方法不同，zip()方法是一个静态的创建操作。创建出来的Flux在角色和他们喜欢的食物之间会完美对齐。从这个合并后的Flux发出的每个条目都是一个Tuple2（一个容纳两个其他对象的容器对象）的实例，其中包含了来自每个源Flux的数据项，并保持着它们发布的顺序。

如果你不想使用Tuple2，而想要使用其他类型，就可以为zip()方法提供一个合并函数来生成你想要的任何对象，合并函数会传入这两个数据项（如图10.8中的弹珠图所示）。

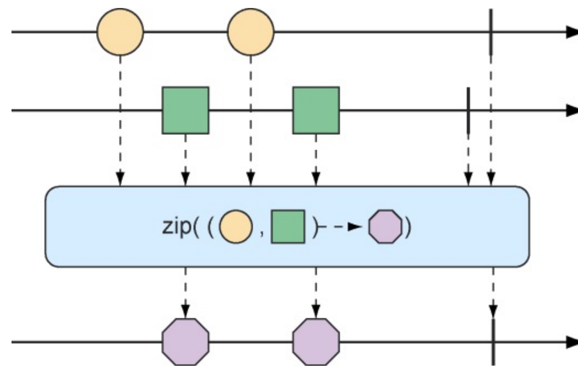


图10.8　zip操作的另一种形式（从每个传入Flux中各取一个元素，然后创建消息对象，并产生这些消息组成的Flux）

例如，下面的测试方法会将角色Flux与食物Flux合并在一起，以便生成一个包含String对象的Flux：

```
@Test
public void zipFluxesToObject() {
  Flux<String> characterFlux = Flux
      .just("Garfield", "Kojak", "Barbossa");
  Flux<String> foodFlux = Flux
      .just("Lasagna", "Lollipops", "Apples");

  Flux<String> zippedFlux =
      Flux.zip(characterFlux, foodFlux, (c, f) -> c + " eats " + f);

  StepVerifier.create(zippedFlux)
        .expectNext("Garfield eats Lasagna")
        .expectNext("Kojak eats Lollipops")
        .expectNext("Barbossa eats Apples")
        .verifyComplete();
}
```

传递给 zip()方法（在这里是一个lambda）的函数只是简单地将两个数据项组装成一个句子，然后通过该合并后的Flux发布出去。

选择第一个反应式类型进行发布

假设我们有两个Flux对象，此时我们不想将它们合并在一起，而是想要创建一个新的Flux，让这个新的Flux从第一个产生值的Flux中发布值。如图10.9所示，first()操作会在两个Flux对象中选择第一个发布值的Flux，并再次发布它的值。

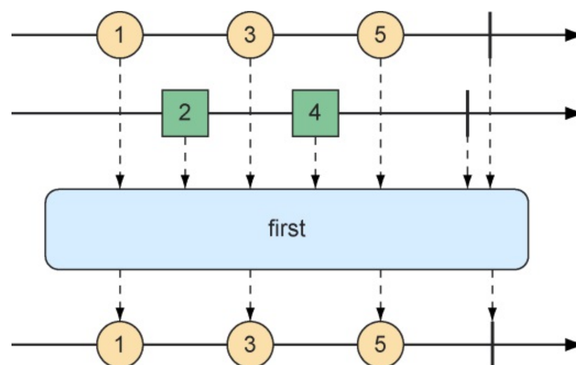


图10.9　first操作将会选择第一个发布消息的Flux并只发布该Flux的值

下面的测试方法创建了一个快速的Flux和一个“缓慢”的Flux（其中“慢”意味着它在被订阅后100毫秒才会发布数据项）。使用first()方法，它将会创建一个新的Flux，这个Flux只会获取第一个源Flux发布的值，并再次发布：

```
@Test
public void firstFlux() {
  Flux<String> slowFlux = Flux.just("tortoise", "snail", "sloth")
        .delaySubscription(Duration.ofMillis(100));
  Flux<String> fastFlux = Flux.just("hare", "cheetah", "squirrel");

  Flux<String> firstFlux = Flux.first(slowFlux, fastFlux);

  StepVerifier.create(firstFlux)
      .expectNext("hare")
      .expectNext("cheetah")
      .expectNext("squirrel")
      .verifyComplete();
}
```

在这种情况下，因为慢速Flux会在快速Flux开始发布之后的100毫秒才发布值，所以新创建的Flux将会简单地忽略慢的Flux，并仅发布来自快速Flux的值。

10.3.3　转换和过滤反应式流

在数据流经一个流时，我们通常需要过滤掉某些值并对其他的值进行处理。在这一节，我们将介绍转换和过滤流经反应式流的数据的操作。

从反应式类型中过滤数据

数据在从Flux流出时，进行过滤的最基本方法之一是简单地忽略第一批指定数目的数据项。skip操作（如图10.10所示）就能完成这样的工作。

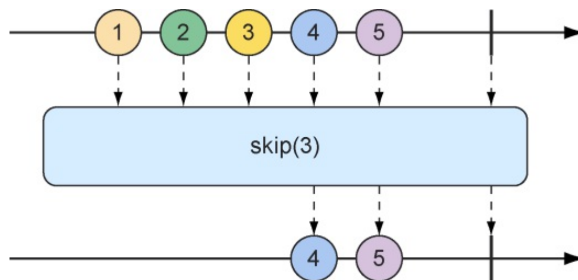


图10.10　skip操作跳过指定数目的消息并将剩下的消息继续在结果Flux上进行传递

针对具有多个数据项的Flux，skip操作将创建一个新的Flux，它会首先跳过指定数量的数据项，然后从源 Flux 中发布剩余的数据项。下面的测试方法展示如何使用skip()方法：

```
@Test
public void skipAFew() {
  Flux<String> skipFlux = Flux.just(
      "one", "two", "skip a few", "ninety nine", "one hundred")
      .skip(3);

  StepVerifier.create(skipFlux)
      .expectNext("ninety nine", "one hundred")
      .verifyComplete();
}
```

在这个场景下，我们有一个具有5个String数据项的Flux。在这个Flux上调用skip(3)方法后会产生一个新的Flux，它会跳过前3个数据项，只发布最后2个数据项。

但是，你可能并不想跳过特定数量的条目，而是想要在一段时间之内跳过所有的第一批数据。这是skip()操作的另一种形式，将会产生一个新Flux，在发布来自源Flux的数据项之前等待指定的一段时间，如图10.11所示。

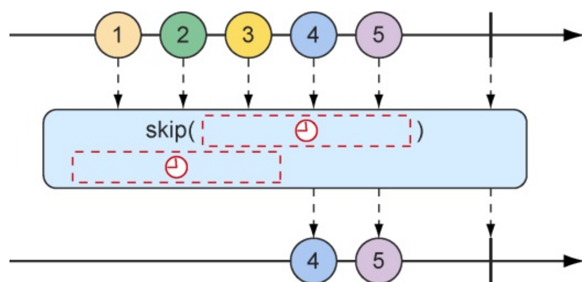


图10.11　skip操作的另一种形式

下面的测试方法使用skip操作创建了一个在发布值之前会等待 4 秒的Flux。因为Flux是基于一个在发布数据项之间有 1 秒延迟的Flux创建的（使用了delayElements()操作），所以它只会发布出最后两个数据项：

```
@Test
public void skipAFewSeconds() {
  Flux<String> skipFlux = Flux.just(
      "one", "two", "skip a few", "ninety nine", "one hundred")
      .delayElements(Duration.ofSeconds(1))
      .skip(Duration.ofSeconds(4));

  StepVerifier.create(skipFlux)
      .expectNext("ninety nine", "one hundred")
      .verifyComplete();
}
```

我们已经看过skip操作的示例，根据对skip操作的描述来看，take可以认为是与skip相反的操作。skip操作会跳过前面几个数据项，而take操

作只发布第一批指定数量的数据项，然后将取消订阅（如图10.12中的弹珠图所示）。

```
@Test
public void take() {
  Flux<String> nationalParkFlux = Flux.just(
          "Yellowstone", "Yosemite", "Grand Canyon",
          "Zion", "Grand Teton")
      .take(3);

  StepVerifier.create(nationalParkFlux)
      .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
      .verifyComplete();
}
```

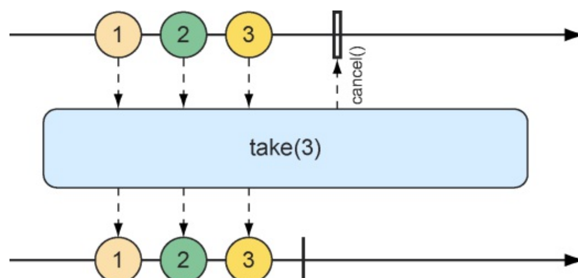


图10.12　take操作只发布传入Flux中前面指定数目的数据项

与skip()方法一样，take()方法也有另一种替代形式，基于间隔时间而不是数据项个数。它将接受并发布与源Flux一样多的数据项，直到某段时间结束，之后Flux将会完成，如图10.13所示。

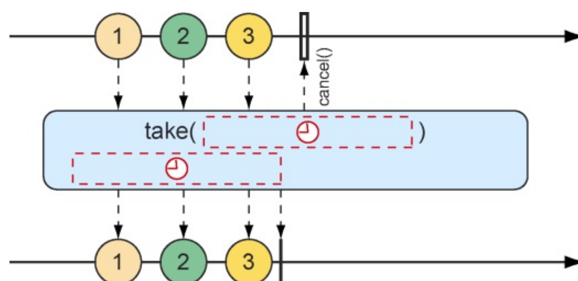


图10.13　take操作的另一种形式（在指定的时间过期之前，一直将消息传递给结果Flux）

下面的测试方法使用take()方法的另一种形式，将会在订阅之后的前3.5秒发布数据条目。

```
@Test
public void take() {
  Flux<String> nationalParkFlux = Flux.just(
          "Yellowstone", "Yosemite", "Grand Canyon",
          "Zion", "Grand Teton")
      .delayElements(Duration.ofSeconds(1))
      .take(Duration.ofMillis(3500));

  StepVerifier.create(nationalParkFlux)
      .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
      .verifyComplete();
}
```

skip操作和take操作都可以被认为是过滤操作，其过滤条件是基于计数或者持续时间的，而Flux值的更通用过滤则是filter操作。

我们需要指定一个Predicate，用于决定数据项是否能通过Flux，filter操作允许我们根据任何条件进行选择性地发布。图10.14中的弹珠图显示了filter操作的工作原理。

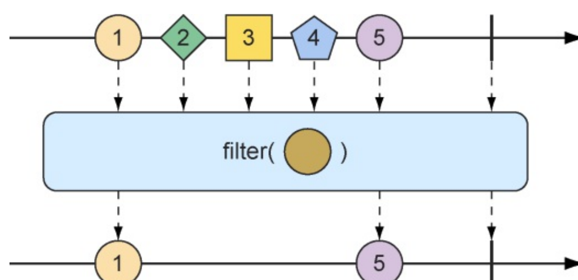


图10.14　可以对传入的Flux进行过滤，这样结果Flux将只会发布满足指定Predicate的消息

要查看filter()的实际效果，可以参考下面的测试方法：

```
@Test
public void filter() {
```

```
  Flux<String> nationalParkFlux = Flux.just(
          "Yellowstone", "Yosemite", "Grand Canyon",
          "Zion", "Grand Teton")
      .filter(np -> !np.contains(" "));
  StepVerifier.create(nationalParkFlux)
      .expectNext("Yellowstone", "Yosemite", "Zion")
      .verifyComplete();
}
```

这里我们将一个只接受不包含空格的字符串的Predicate作为lambda传给了filter()方法，因此在结果Flux中"Grand Canyon"和"Grand Teton"被过滤掉了。

我们还可能想要过滤掉已经接收过的数据条目，可以采用distinct操作（如图10.15所示），形成的Flux将只会发布源Flux中尚未发布过的数据项。

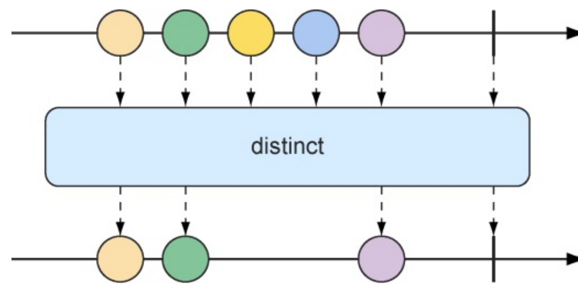


图10.15　distinct操作将会过滤掉重复的消息

在下面的测试中，调用distinct()方法产生的Flux只会发布不同的String值：

```
@Test
public void distinct() {
  Flux<String> animalFlux = Flux.just(
      "dog", "cat", "bird", "dog", "bird", "anteater")
      .distinct();

  StepVerifier.create(animalFlux)
```

```
        .expectNext("dog", "cat", "bird", "anteater")
        .verifyComplete();
}
```

虽然"dog"和"bird"从源Flux中都发布了两次，但是在调用distinct()方法产生的结果Flux中，它们只被发布了一次。

映射反应式数据

在Flux或Mono上最常见的操作之一就是将已发布的数据项转换为其他的形式或类型。Reactor的反应式类型（Flux和Mono）为此提供了map和flatMap操作。

map操作会创建一个新的Flux，只是在重新发布它所接收的每个对象之前会执行给定Function指定的转换。map操作的工作原理如图10.16所示。

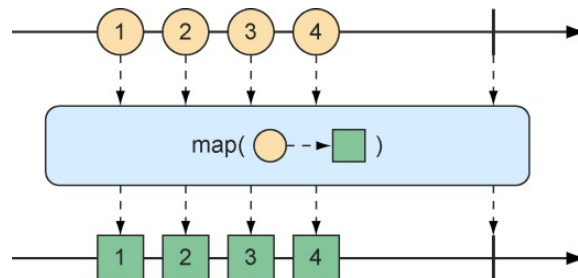


图10.16　map 操作将传入的消息转换为结果流上的新消息

在下面的test()方法中，包含代表篮球运动员名字的String值的Flux被转换为一个包含Player对象的新Flux。

```
@Test
public void map() {
  Flux<Player> playerFlux = Flux
```

```
    .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
    .map(n -> {
      String[] split = n.split("\\s");
      return new Player(split[0], split[1]);
    });

  StepVerifier.create(playerFlux)
      .expectNext(new Player("Michael", "Jordan"))
      .expectNext(new Player("Scottie", "Pippen"))
      .expectNext(new Player("Steve", "Kerr"))
      .verifyComplete();
}
```

以lambda形式传递给map()方法的函数会将传入的String值按照空格进行拆分，并使用生成的String数组来创建Player对象。使用just()方法创建的Flux包含了String对象，但是map()方法产生的Flux包含了Player对象。

其中重要的一点是：在每个数据项被源Flux发布时，map操作是同步执行的，如果你想要异步地转换过程，那么你应该考虑使用flatMap操作。

对于flatMap操作，我们可能需要一些思考和练习才能完全掌握。如图10.17所示，flatMap并不像map操作那样简单地将一个对象转换到另一个对象，而是将对象转换为新的Mono或Flux。结果形成的Mono或Flux会扁平化为新的Flux。当与subscribeOn()方法结合使用时，flatMap操作可以释放Reactor反应式的异步能力。

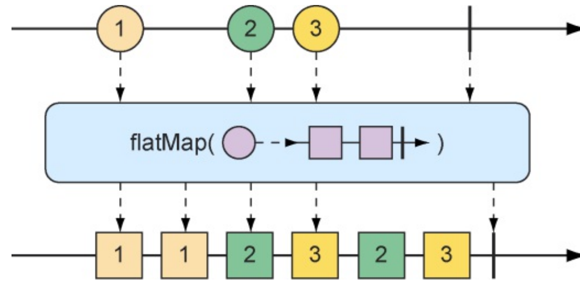


图10.17　flatMap操作使用一个中间的Flux来实现异步转换

下面的测试方法展示如何使用flatMap()方法和subscribeOn()方法：

```
@Test
public void flatMap() {
  Flux<Player> playerFlux = Flux
    .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
    .flatMap(n -> Mono.just(n)
        .map(p -> {
            String[] split = p.split("\\s");
            return new Player(split[0], split[1]);
          })
        .subscribeOn(Schedulers.parallel())
      );

  List<Player> playerList = Arrays.asList(
      new Player("Michael", "Jordan"),
      new Player("Scottie", "Pippen"),
      new Player("Steve", "Kerr"));

  StepVerifier.create(playerFlux)
      .expectNextMatches(p -> playerList.contains(p))
      .expectNextMatches(p -> playerList.contains(p))
      .expectNextMatches(p -> playerList.contains(p))
      .verifyComplete();
}
```

需要注意的是，我们为flatMap()方法指定了一个lambda形式的函数，传入的String将会转换为一个Mono类型的String，然后在这个Mono上通过map()方法将字符串转换为一个Player。

如果到此为止，那么产生的Flux将同样包含Player对象，与使用map()方法的例子相同，顺序同步地生成。但是我们对Mono做的最后一个动作就是调用subscribeOn()方法，它声明每个订阅都应该在并行线程中进行，因此可以异步并行地执行多个String对象的转换操作。

尽管subscribeOn()方法的命名与subscribe()方法类似，但是它们的含义却完全不同。 subscribe()方法是一个动词，订阅并驱动反应式流；而subscribeOn()方法则更具描述性，指定了如何并发地处理订阅。Reactor本身并不强制使用特定的并发模型，通过subscribeOn()方法，我们可以使用Schedulers中的任意一个静态方法来指定并发模型。在这个例子中，我们使用了parallel()方法，使用来自固定线程池（大小与CPU核心数量相同）的工作线程。Schedulers支持多种并发模型，如表10.1所示。

表**10.1　Schedulers**支持的并发模型

Schedulers方法	描述
.immediate()	在当前的线程中执行订阅
.single()	在一个单一的、可重用的线程中执行订阅。对所有的调用者重用相同的线程
.newSingle()	针对每个调用，使用专用的线程执行订阅
	在从无界弹性线程池中拉取的工作者线程中执行订阅。 根据需要创建

.elastic()	新的工作线程，并销毁空闲的工作者线程（默认情况下，会在空闲60秒后销毁）
.parallel()	在从一个固定大小的线程池中拉取的工作者线程中执行订阅，该线程池的大小和CPU的核心数一致

使用flatMap()和subscribeOn()的好处是：我们可以在多个并行线程之间拆分工作，从而增加流的吞吐量。因为工作是并行完成的，无法保证哪项工作首先完成，所以结果Flux中数据项的发布顺序是未知的。因此，StepVerifier只能验证发出的每个数据项是否存在于预期Player对象列表中，并且在Flux完成之前会有3个这样的数据项。

在反应式流上缓存数据

在处理流经Flux的数据时，你可能会发现将数据流拆分为小块会带来一定的收益。如图10.18所示的buffer操作可以帮助你解决这个问题。

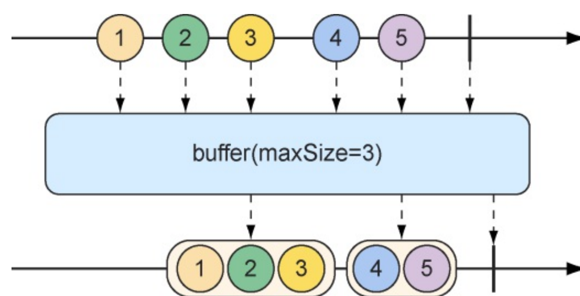


图10.18　buffer操作会产生一个新的包含列表Flux（具备最大长度限制的列表，包含从传入的Flux中收集来的数据）

我们给定一个包含多个String值的Flux，其中每个值代表一种水果的名称，我们可以创建一个新的包含List 集合的Flux，其中每个List只有

不超过指定数量的元素：

```
@Test
public void buffer() {
  Flux<String> fruitFlux = Flux.just(
      "apple", "orange", "banana", "kiwi", "strawberry");

  Flux<List<String>> bufferedFlux = fruitFlux.buffer(3);

  StepVerifier
      .create(bufferedFlux)
      .expectNext(Arrays.asList("apple", "orange", "banana"))
      .expectNext(Arrays.asList("kiwi", "strawberry"))
      .verifyComplete();
}
```

在这种情况下，String元素的Flux被缓冲到一个新的包含List集合的Flux中，其中每个集合不超过3个条目。因此，发出5个String值的原始Flux将会被转换为一个新的Flux，它会发出两个List集合，其中一个包含3个水果，而另一个包含2个水果。

这有什么意义呢？将反应式的Flux缓冲到非反应式的Flux中看起来适得其反，但是，当组合使用buffer()方法和flatMap()方法时，我们可以对每个List集合进行并行处理。

```
Flux.just(
    "apple", "orange", "banana", "kiwi", "strawberry")
    .buffer(3)
    .flatMap(x ->
      Flux.fromIterable(x)
        .map(y -> y.toUpperCase())
        .subscribeOn(Schedulers.parallel())
        .log()
    ).subscribe();
```

在这个新例子中，我们仍然将5个String值的Flux缓冲到一个新的包

含List的Flux中，但是这次将flatMap()应用于包含List集合的Flux。这将获取每个List缓冲区，并为其中的元素创建一个新的Flux，然后对其应用map操作。因此，每个List缓冲区都会在各个线程中执行进一步并行处理。

为了观察实际的效果，在代码中还包括了一个log()操作，它用于每个子Flux。 log()操作记录了所有的反应式事件，以便于观察实际发生了什么事情。在日志中将会记录如下的条目（为简洁起见，删除了时间戳的部分）：

```
[main] INFO reactor.Flux.SubscribeOn.1 -
                  onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.1 - request(32)
[main] INFO reactor.Flux.SubscribeOn.2 -
                  onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.2 - request(32)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(APPLE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(KIWI)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(ORANGE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(STRAWBERRY)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(BANANA)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onComplete()
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onComplete()
```

如同日志记录所清晰展示的，第一个缓冲区（apple、orange和banana）中的水果在parallel-1线程中处理；与此同时，第二个缓冲区（kiwi和strawberry）中的水果在parallel-2 线程中处理。从缓冲区的日志记录交织在一起的事实可以明显地看出，对两个缓冲区的处理是并行执行的。

如果由于某些原因需要将Flux发布的所有数据项都收集到一个List

中，那么可以使用不带参数的buffer()方法：

```
Flux<List<String>> bufferedFlux = fruitFlux.buffer();
```

这将会产生一个新的Flux。这个Flux将会发布一个List，其中包含源Flux发布的所有数据项。我们可以使用collectList操作实现相同的功能，如图10.19中的弹珠图所示。

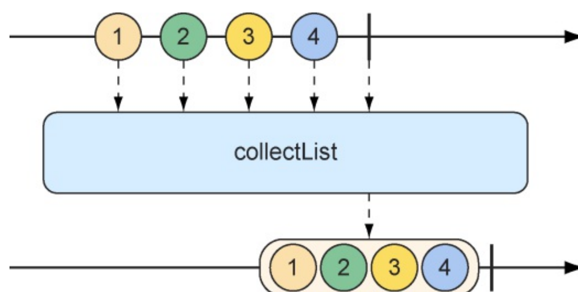


图10.19　collectList操作将产生一个包含传入Flux发布的所有消息的Mono

collectList()方法会产生一个发布List的Mono，而不是发布List的Flux。下面的测试方法展示了它的用法：

```
@Test
public void collectList() {
  Flux<String> fruitFlux = Flux.just(
      "apple", "orange", "banana", "kiwi", "strawberry");

  Mono<List<String>> fruitListMono = fruitFlux.collectList();

  StepVerifier
      .create(fruitListMono)
      .expectNext(Arrays.asList(
          "apple", "orange", "banana", "kiwi", "strawberry"))
      .verifyComplete();
}
```

一种更加有趣的收集Flux发出的数据项的方法是将它们收集到Map

中。如图10.20所示，collectMap 操作将会产生一个发布Map的Mono，这个Map中填充了由给定Function计算key值所生成的条目。

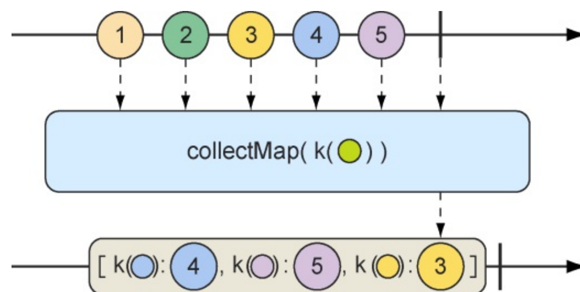


图10.20　collectMap操作将会产生一个Mono（包含了由传入Flux所发出的消息产生的Map，这个Map的key是从传入消息的某些特征衍生而来的）

要查看collectMap()的效果，请参考下面的测试方法：

```
@Test
public void collectMap() {
  Flux<String> animalFlux = Flux.just(
      "aardvark", "elephant", "koala", "eagle", "kangaroo");

  Mono<Map<Character, String>> animalMapMono =
      animalFlux.collectMap(a -> a.charAt(0));

  StepVerifier
      .create(animalMapMono)
      .expectNextMatches(map -> {
        return
            map.size() == 3 &&
            map.get('a').equals("aardvark") &&
            map.get('e').equals("eagle") &&
            map.get('k').equals("kangaroo");
      })
      .verifyComplete();
}
```

源Flux会发布一些动物的名字。基于这个Flux，我们使用collectMap操作创建了一个发布Map的新Mono，其中key由动物名称的首字母确

定，而值则为动物名称本身。 如果两个动物名称以相同的字母开头（如elephant和eagle，或者 koala 和kangaroo），那么流经该流的最后一个条目将会覆盖先前的条目。

10.3.4　在反应式类型上执行逻辑操作

有时候我们想要知道由Mono或者Flux发布的条目是否满足某些条件，那么all()和any()方法可以实现这样的逻辑。图10.21和图10.22展示了all()和any()的工作方式。

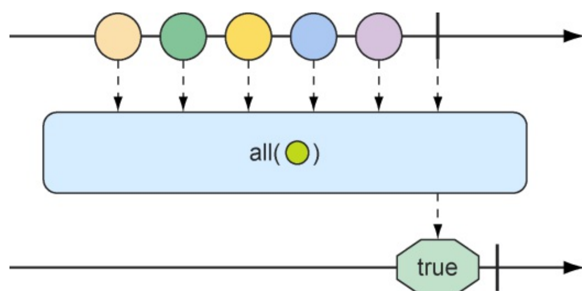


图10.21　可以使用all()方法来确保Flux中的所有消息都满足某些条件

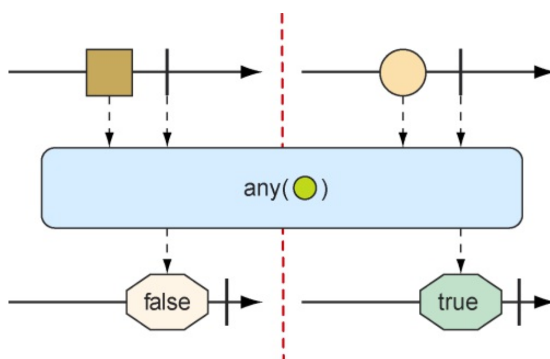


图10.22　可以使用any()方法来确保Flux中至少有一个消息满足某些条件

假设我们想知道Flux发布的每个String中是否都包含了字母a或字母

k，那么下面的测试将使用all()方法来检查这个条件：

```
@Test
public void all() {
  Flux<String> animalFlux = Flux.just(
      "aardvark", "elephant", "koala", "eagle", "kangaroo");

  Mono<Boolean> hasAMono = animalFlux.all(a -> a.contains("a"));
  StepVerifier.create(hasAMono)
    .expectNext(true)
    .verifyComplete();

  Mono<Boolean> hasKMono = animalFlux.all(a -> a.contains("k"));
  StepVerifier.create(hasKMono)
    .expectNext(false)
    .verifyComplete();
}
```

在第一个StepVerifier中，我们检查了字母a。all()方法应用于源Flux，会产生布尔类型的Mono。在本例中，所有动物名称都包含了字母a，所以从生成的Mono中会发布 true。但是在第二个StepVerifier中，产生的Mono将会发出false，因为并非所有动物名称都包含字母k。

如果至少有一个元素匹配条件即可，而不是要求所有元素均满足条件，那么 any()就是我们所需的方法。下面这个新的测试用例使用any()来检查字母t和z：

```
@Test
public void any() {
  Flux<String> animalFlux = Flux.just(
      "aardvark", "elephant", "koala", "eagle", "kangaroo");

  Mono<Boolean> hasAMono = animalFlux.any(a -> a.contains("t"));

  StepVerifier.create(hasAMono)
    .expectNext(true)
    .verifyComplete();
```

```
  Mono<Boolean> hasZMono = animalFlux.any(a -> a.contains("z"));
  StepVerifier.create(hasZMono)
    .expectNext(false)
    .verifyComplete();
}
```

在第一个StepVerifier中，我们会看到生成的Mono发布了true，因为至少有一种动物的名称含有字母t（尤其是elephant）。而在第二种情况下，生成的Mono发布了false，因为没有任何一种动物的名称包含字母z。

10.4 小结

- 反应式编程会涉及创建数据流经的处理管道。
- 反应式流规范定义了4种类型：Publisher、Subscriber、Subscription和Processor（它是Publisher和Subscriber的组合）。
- Reactor项目实现了反应式流规范，将反应式流的定义抽象为两个主要的类型，即Flux和Mono，并为每种类型都提供数百个操作。
- Spring 5利用Reactor提供了反应式控制器、repository、REST客户端以及其他反应式框架的支持。

第11章　开发反应式API

本章内容：

- 使用Spring WebFlux
- 编写和测试反应式的控制器以及客户端
- 消费REST API
- 保护反应式Web应用

我们已经了解了反应式编程和Reactor项目，现在可以开始在Spring应用程序中使用这些技术了。在本章中，我们将利用Spring 5的反应式编程模型重新讨论在第6章中编写的控制器。

具体来讲，我们将一起探讨Spring 5中新添加的反应式Web框架——Spring WebFlux。我们很快会发现，Spring WebFlux与Spring MVC非常相似，这样使得它非常易于使用，我们已经掌握的如何在Spring中构建REST API的知识依然有用。

11.1　使用Spring WebFlux

传统的基于Servlet的Web框架，如Spring MVC，在本质上都是阻塞和多线程的，每个连接都会使用一个线程。在请求处理的时候，会在线程池中拉取一个工作者（worker）线程来对请求进行处理。同时，请求线程是阻塞的，直到工作者线程提示它已经完成为止。

这样带来的后果就是阻塞式Web框架在大量请求下无法有效地扩展。缓慢的工作者线程所带来的延迟会使情况变得更糟，因为它将花费更长的时间才能将工作者线程送回池中，准备处理另一个请求。在某些场景中，这种设计完全可以接受。事实上，在很大程度上这就是十多年来大多数Web应用程序的开发方式，但是时代在改变。

这些Web应用程序的客户端以前是偶尔浏览网站的人们，而现在这些人会频繁消费内容而且会使用与HTTP API协作的应用程序。如今，物联网（甚至不需要人类）产生了汽车、喷气式发动机和其他非传统的客户端，它们会持续地和Web API交换数据。随着消费Web应用的客户端越来越多，可扩展性比以往任何时候都更加重要。

异步的Web框架能够以更少的线程获得更高的可扩展性，通常它们只需要与CPU核心数量相同的线程。通过使用所谓的事件轮询（event looping）机制（如图11.1所示），这些框架能够用一个线程处理很多请求，这样每次连接的成本会更低。

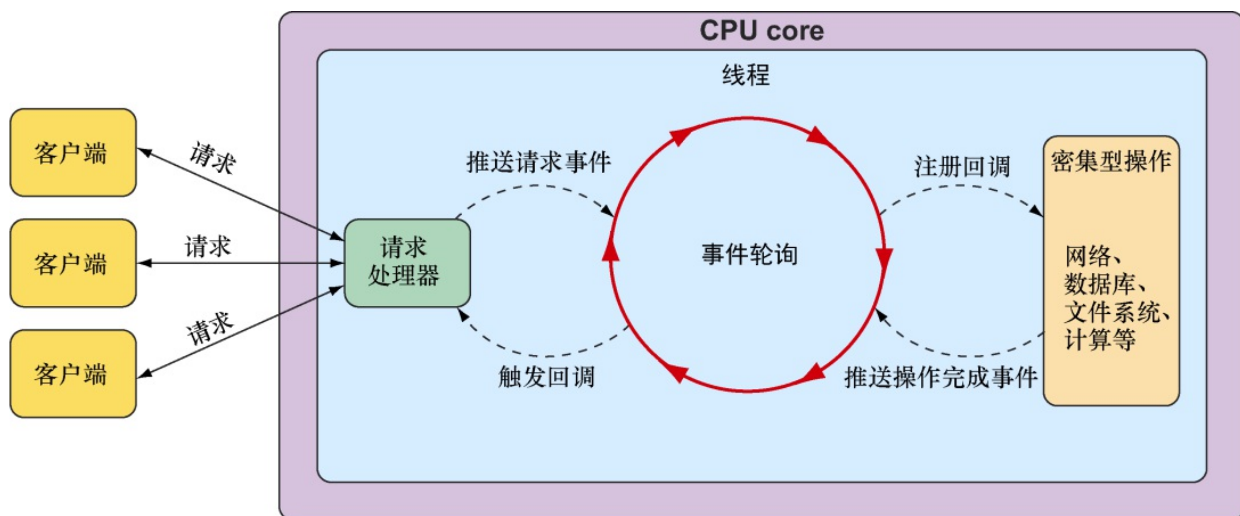


图11.1　异步Web框架借助事件轮询机制能够以更少的线程处理更多的请求

在事件轮询中，所有事情都是以事件的方式来进行处理的，包括请求以及密集型操作（如数据库和网络操作）的回调。当需要执行成本高昂的操作时，事件轮询会为该操作注册一个回调，这样操作可以并行执行，而事件轮询则会继续处理其他的事件。

当操作完成时，事件轮询机制会将其作为一个事件，这一点与请求是相同的。这样达到的效果就是，在面临大量负载的时候，异步Web框架能够以更少的线程实现更好的可扩展性，这样会减少线程管理的开销。

Spring 5引入了一个非阻塞、异步的Web框架，该框架在很大程度上是基于Reactor项目的，能够解决Web应用和API中对更好的可扩展性的需求。接下来我们看一下Spring WebFlux：面向Spring的反应式Web框架。

11.1.1　Spring WebFlux简介

当Spring团队思考如何向Web层添加反应式编程模型时，如果不在Spring MVC中做大量工作，显然很难实现这一点。这会在代码中产生分支以决定是否要以反应式的方式来处理请求。如果这样做，本质上就是将两个Web框架打包成一个，依靠if语句来区分反应式和非反应式。

与其将反应式编程模型硬塞进Spring MVC中，还不如创建一个单独的反应式Web框架，并尽可能多地借鉴Spring MVC。这样，Spring WebFlux就应运而生了。Spring 5定义的完整Web开发技术栈如图11.2所示。

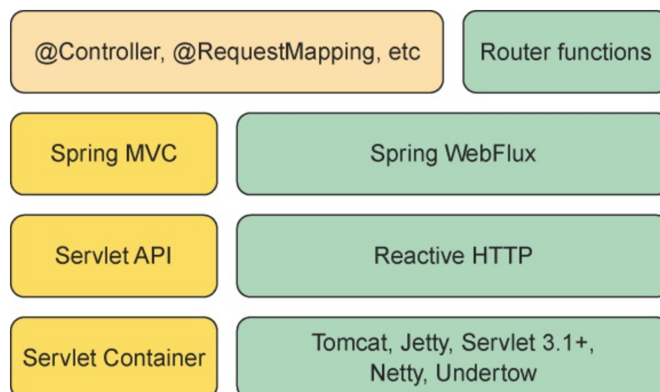


图11.2　Spring 5通过名为WebFlux的新Web框架来支持反应式Web应用

在图11.2的左侧，我们会看到Spring MVC技术栈，这是Spring框架2.5版本就引入的。Spring MVC（在第2章和第6章已经进行了讨论）建立在Java Servlet API之上，因此需要Servlet容器（比如Tomcat）才能执行。

与之不同，Spring WebFlux（在图11.2的右侧，和Spring MVC系出同门，并且很多核心组件都是公用的）并不会绑定Servlet API，所以它

构建在Reactive HTTP API之上，这个API与Servlet API具有相同的功能，只不过是采用了反应式的方式。因为Spring WebFlux没有与Servlet API耦合，所以它的运行并不需要Servlet容器。它可以运行在任意非阻塞Web容器中，包括Netty、Undertow、Tomcat、Jetty或任意Servlet 3.1及以上的容器。

在图11.2中，最值得注意的是左上角，它代表了Spring MVC和Spring WebFlux公用的组件，主要用来定义控制器的注解。因为Spring MVC和Spring WebFlux会使用相同的注解，所以Spring WebFlux与Spring MVC在很多方面并没有区别。

右上角的方框表示另一种编程模型，它使用函数式编程范式来定义控制器，而不是使用注解。在11.2节中，我们将更多地讨论Spring的函数式Web编程模型。

Spring MVC和Spring WebFlux之间最显著的区别在于，我们要将哪个依赖项添加到构建文件中。在使用Spring WebFlux时，我们需要添加Spring Boot WebFlux starter依赖项，而不是标准的Web starter（例如，spring-boot-starter-web）。在项目的pom.xml文件中，如下所示：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

注意：与Spring Boot的大多数starter依赖类似，这个starter也可以在Initializr中通过选中Reactive Web复选框添加到项目中。

使用WebFlux有一个很有意思的副作用，即WebFlux的默认嵌入式服务器是Netty而不是Tomcat。Netty是一个异步、事件驱动的服务器，非常适合Spring WebFlux这样的反应式Web框架。

除了使用不同的starter依赖之外，Spring WebFlux的控制器方法要接受和返回反应式类型，如Mono和Flux，而不是领域类型和集合。Spring WebFlux控制器也能处理RxJava类型，如Observable、Single和Completable。

反应式Spring MVC

尽管Spring WebFlux控制器通常会返回Mono和Flux，但是这并不意味着Spring MVC无法体验反应式类型的乐趣。如果你愿意，那么Spring MVC也可以返回Mono和Flux。

这里的区别在于，这些类型会如何被使用。Spring WebFlux是真正的反应式Web框架，允许在事件轮询中处理请求；而Spring MVC是基于Servlet的，依赖于多线程来处理多个请求。

接下来，我们让Spring WebFlux运行起来，借助Spring WebFlux重新编写Taco Cloud的API控制器。

11.1.2　编写反应式控制器

你可能还记得在第6章中我们为Taco Cloud的REST API创建了一些控制器，这些控制器中包含请求处理方法，这些方法会以领域类型（如Order和Taco）或领域类型集合的方式处理输入和输出。作为提醒，我们看一下在第6章所编写的DesignTacoController片段：

```
@RestController
@RequestMapping(path="/design",
                produces="application/json")
@CrossOrigin(origins="*")
public class DesignTacoController {

...

  @GetMapping("/recent")
  public Iterable<Taco> recentTacos() {
    PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
    return tacoRepo.findAll(page).getContent();
  }

...

}
```

按照以上的编写形式，recentTacos()控制器会处理对“/design/recent”的HTTP GET请求，返回最近创建的taco列表。具体来讲，它会返回Taco类型的Iterable。这主要是因为repository的findAll()方法返回的就是该类型，或者更准确地说，这个结果来自findAll()方法所返回的Page对象的getContent()方法。

这样能够很好地运行，但是Iterable并不是反应式类型。我们不能对它使用任何反应式操作，也不能让框架将它视为反应式类型，从而将工作切分到多个线程中。我们希望recentTacos()方法能够返回Flux<Taco>。

这里有一个简单但效果有限的方案：重写recentTacos()，将Iterable转换为Flux。而且，在重写的时候，我们可以去掉分页代码，将其替换为调用Flux的take()：

```
@GetMapping("/recent")
public Flux<Taco> recentTacos() {
  return Flux.fromIterable(tacoRepo.findAll()).take(12);
}
```

借助Flux.fromIterable()，我们可以将Iterable<Taco>转换为Flux<Taco>。既然我们可以使用Flux了，那么就能使用take操作将Flux返回的值限制为最多12个Taco对象。不仅代码更加简洁，而且我们能够处理反应式的Flux，而不是简单的Iterable。

到目前为止，我们编写反应式代码一切都很顺利。如果repository一开始就给我们一个Flux那就更好了，就没有必要进行转换了。如果能够实现这一点，那么recentTacos()将会写成如下形式：

```
@GetMapping("/recent")
public Flux<Taco> recentTacos() {
  return tacoRepo.findAll().take(12);
}
```

这样就更好了！在理想情况下，反应式控制器将会位于反应式端到端栈的顶部，这个栈包括了控制器、repository、数据库以及在它们之间可能还会包含的服务。这样的端到端反应式栈如图11.3所示。

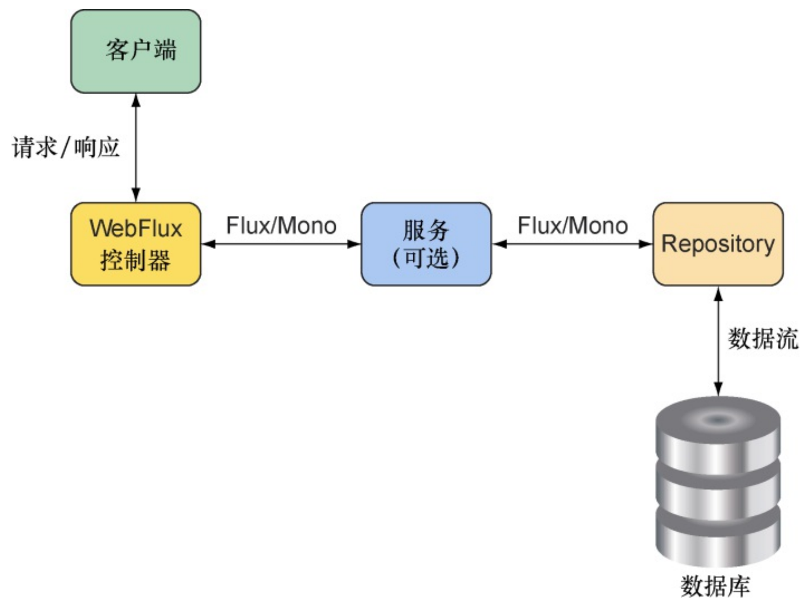


图11.3　它应该成为完整的端到端反应式栈的一部分（为了最大化反应式Web框架的收益）

这样的端到端技术栈要求repository返回Flux，而不是Iterable。在第12章中，我们将会详细研究如何编写反应式repository，但是反应式TacoRepository大致会如下所示：

```
public interface TacoRepository
         extends ReactiveCrudRepository<Taco, Long> {
}
```

此时，最需要注意的事情在于除了使用Flux来替换Iterable以及如何获取Flux之外，定义反应式WebFlux控制器的编程模型与非反应式Spring MVC控制器并没有什么差异。它们都使用了@RestController注解并且都在类级别使用了@RequestMapping。它们都有在方法级别使用@GetMapping注解的请求处理函数。真正重要的是处理器方法返回了什么类型。

另外值得注意的是，尽管我们从repository得到了Flux<Taco>，但是

我们直接将它返回了，并没有调用subscribe()。框架将会为我们调用subscribe()。这意味着当处理“/design/recent”请求的时候，recentTacos()方法会被调用，在数据真正从数据库取出之前它就能立即返回。

返回单个值

作为另外一个样例，我们思考一下在第6章中编写的DesignTacoController的tacoById()方法：

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return optTaco.get();
  }
  return null;
}
```

在这里，该方法处理对“/design/{id}”的GET请求并返回单个Taco对象。因为repository的findById()返回的是Optional，所以我们必须编写一些烦琐的代码处理它。如果findById()返回的是Mono<Taco>，而不是Optional<Taco>，那么我们可以按照如下的方式重写控制器的tacoById()：

```
@GetMapping("/{id}")
public Mono<Taco> tacoById(@PathVariable("id") Long id) {
  return tacoRepo.findById(id);
}
```

这样看上去简单多了。更重要的是，通过返回Mono<Taco>来替代Taco，我们能够让Spring WebFlux以反应式的方式处理响应。这样做的

结果就是在面临高负载的时候我们的API能够更好地进行扩展。

使用**RxJava**类型

值得一提的是，在使用Spring WebFlux时，虽然Flux和Mono是自然而然的选择，但是我们也可以使用像Observable和Single这样的RxJava类型。例如，假设在DesignTacoController和后端repository之间有一个服务，处理的是RxJava类型，那么recentTacos()方法可以编写为：

```
@GetMapping("/recent")
public Observable<Taco> recentTacos() {
  return tacoService.getRecentTacos();
}
```

类似的，tacoById()方法可以编写成处理RxJava Single类型，而不是Mono类型：

```
@GetMapping("/{id}")
public Single<Taco> tacoById(@PathVariable("id") Long id) {
  return tacoService.lookupTaco(id);
}
```

除此之外，Spring WebFlux控制器方法还可以返回RxJava的Completable，后者等价于Reactor中的Mono<Void>。WebFlux也可以返回Flowable，替换Observable或Reactor的Flux。

实现输入的反应式

到目前为止，我们只关心了控制器方法返回什么样的反应式类型。但是，借助Spring WebFlux，我们还可以接受Mono或Flux作为处理器方

法的输入。为了阐述这一点，请思考DesignTacoController中原始的postTaco()实现：

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
  return tacoRepo.save(taco);
}
```

按照原始的编写方式，postTaco()不仅会返回一个简单的Taco对象，还会接受一个Taco，这个对象绑定了请求体中的内容。这意味着在请求载荷完成解析并初始化为Taco对象之前，postTaco()方法是不会被调用的。这同时也意味着，在对repository的save()方法的阻塞调用返回之前，postTaco()是不能返回的。简而言之，这个请求阻塞了两次：在进入postTaco()的时候以及在postTaco()调用的过程中。通过为postTaco()添加一些反应式代码，我们能够将它变成完全非阻塞的请求处理方法：

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Taco> postTaco(@RequestBody Mono<Taco> tacoMono) {
  return tacoRepo.saveAll(tacoMono).next();
}
```

在这里，postTaco()接受一个Mono<Taco>并调用了repository的saveAll()方法。我们将会在第12章看到这个repository能够接受反应式流Publisher的任意实现，包括Mono或Flux。saveAll()方法返回了一个Flux<Taco>，但我们想要的是Mono。我们知道该Flux最多只能发布一个Taco，所以调用next()方法获取postTaco()方法要返回的Mono<Taco>。

通过接受Mono<Taco>作为输入，方法会立即调用，不用等待从请

求体中解析生成Taco。另外，repository也是反应式的，它接受一个Mono并立即返回Flux<Taco>，所以我们调用Flux的next()来获取最终的Mono<Taco>。方法在请求真正处理之前就能返回。

Spring WebFlux是一个非常棒的Spring MVC替代方案，提供了与Spring MVC相同的开发模型来编写反应式Web应用。其实Spring 5还有另外一项技巧，下面让我们看看如何使用Spring 5的新函数式编程风格创建反应式API。

11.2 定义函数式请求处理器

Spring MVC基于注解的编程模型从Spring 2.5就存在了，而且这种模型非常流行，但是它也有一些缺点。

首先，所有基于注解的编程方式都会存在注解该做什么以及注解如何做之间的割裂。注解本身定义了该做什么，而具体如何去做则是在框架代码的其他部分定义的。如果想要进行自定义或扩展，编程模型就会变得很复杂，因为这样的变更需要修改注解之外的代码。除此之外，这种代码的调试也是比较麻烦的，因为我们无法在注解上设置断点。

其次，随着Spring变得越来越流行，很多熟悉其他语言和框架的Spring新手会觉得基于注解的Spring MVC（和WebFlux）与他们之前掌握的知识有很大的差异。作为注解式WebFlux的一种替代方案，Spring 5引入了一个新的函数式编程模型，用来定义反应式API。

这个新的编程模型使用起来更像一个库，而不是一个框架，能够让

我们在不使用注解的情况下将请求映射到处理器代码中。使用Spring的函数式编程模型编写API会涉及4个主要的类型：

- RequestPredicate：声明要处理的请求类型。
- RouterFunction：声明如何将请求路由到处理器代码中。
- ServerRequest：代表一个HTTP请求，包括对请求头和请求体的访问。
- ServerResponse：代表一个HTTP响应，包括响应头和响应体信息。

下面是一个将所有类型组合在一起的Hello World样例：

```
package demo;
import static org.springframework.web.
                   reactive.function.server.RequestPredicates.GET;
import static org.springframework.web.
                   reactive.function.server.RouterFunctions.route;
import static org.springframework.web.
                   reactive.function.server.ServerResponse.ok;
import static reactor.core.publisher.Mono.just;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;

@Configuration
public class RouterFunctionConfig {

  @Bean
  public RouterFunction<?> helloRouterFunction() {
    return route(GET("/hello"),
        request -> ok().body(just("Hello World!"), String.class));
  }

}
```

我们需要注意的第一件事情就是，在这里静态导入了一些辅助类，可以使用它们来创建前文所述的函数式类型。我们还以静态方式导入了

Mono，从而能够让剩余的代码更易于阅读和理解。

在这个@Configuration类中，我们有一个类型为RouterFunction<?>的@Bean方法。按照前文所述，RouterFunction能够声明一个或多个RequestPredicate对象和处理与之匹配的请求的函数之间的映射关系。

RouterFunctions的route()方法接受两个参数：RequestPredicate以及处理与之匹配的请求的函数。在本例中，RequestPredicates的GET()方法声明一个RequestPredicate，后者会匹配针对“/hello”的HTTP GET请求。

至于处理器函数，它写成了lambda表达式的形式，当然它也可以使用方法引用。尽管这里没有显式声明，但是处理器lambda表达式会接受一个ServerRequest作为参数。它通过ServerResponse的ok()方法和BodyBuilder的body()方法返回了一个ServerResponse。BodyBuilder对象是由ok()所返回的。这样的话，就会创建出状态码为HTTP 200 (OK)并且响应体载荷为“Hello World!”的响应。

按照这种编写形式，helloRouterFunction()方法所声明的RouterFunction只能处理一种类型的请求。如果想要处理不同类型的请求，那么我们没有必要编写另外一个@Bean（当然你也可以这样做），仅需调用andRoute()来声明另一个RequestPredicate到函数的映射。例如，为“/bye”的GET请求添加一个处理器：

```
@Bean
public RouterFunction<?> helloRouterFunction() {
  return route(GET("/hello"),
      request -> ok().body(just("Hello World!"), String.class))
    .andRoute(GET("/bye"),
      request -> ok().body(just("See ya!"), String.class));
```

```
}
```

Hello World这种级别的样例只能用来简单体验一些新东西。接下来，我们进一步看一下如何使用Spring的函数式Web编程模型处理接近真实场景的请求。

为了阐述如何在真实应用中使用函数式编程模型，我们会使用函数式风格重新实现DesignTacoController的功能。如下的配置类是DesignTacoController的函数式实现：

```
@Configuration
public class RouterFunctionConfig {

  @Autowired
  private TacoRepository tacoRepo;

  @Bean
  public RouterFunction<?> routerFunction() {
    return route(GET("/design/taco"), this::recents)
      .andRoute(POST("/design"), this::postTaco);
  }

  public Mono<ServerResponse> recents(ServerRequest request) {
    return ServerResponse.ok()
        .body(tacoRepo.findAll().take(12), Taco.class);
  }

  public Mono<ServerResponse> postTaco(ServerRequest request) {
    Mono<Taco> taco = request.bodyToMono(Taco.class);
    Mono<Taco> savedTaco = tacoRepo.save(taco);
    return ServerResponse
        .created(URI.create(
             "http://localhost:8080/design/taco/" +
             savedTaco.getId()))
        .body(savedTaco, Taco.class);
  }
}
```

我们可以看到，routerFunction()方法声明了一个RouterFunction<?>

bean，这与Hello World样例类似。但是，它们之间的差异在于要处理什么类型的请求以及如何处理。在本例中，我们创建的RouterFunction处理针对“/design/taco”的GET请求以及针对“/design”的POST请求。

更明显的差异在于，路由是由方法引用处理的。如果RouterFunction背后的行为相对简单和简洁，那么lambda是很不错的选择。在很多场景下，最好将功能抽取到一个单独的方法中（甚至抽取到一个独立类的方法中），以便于保持代码的可读性。

就我们的需求而言，针对“/design/taco”的GET请求将由recents()方法来处理。它使用注入的TacoRepository得到一个Flux<Taco>，然后从中得到12个条目。针对“/design”的POST请求会由postTaco()方法来处理，它会从传入的ServerRequest中抽取Mono<Taco>。postTaco()使用TacoRepository方法来进行保存，随后使用save()返回的Mono<Taco>作为响应。

11.3　测试反应式控制器

在反应式控制器的测试方面，Spring 5并没有置我们于不顾。实际上，Spring 5引入了WebTestClient。这是一个新的测试工具类，让Spring WebFlux编写的反应式控制器的测试变得非常容易。为了了解如何使用WebTestClient编写测试，我们首先使用它测试11.1.2小节中编写的DesignTacoController中的recentTacos()方法。

11.3.1　测试GET请求

对于recentTacos()方法，我们想断言如果针对“/design/recent”路径发送HTTP GET请求，那么将会得到JSON载荷的响应并且taco的数量不会超过12个。程序清单 11.1中的测试类将会是一个很好的起点。

程序清单11.1　使用WebTestClient测试DesignTacoController

```
package tacos;

import static org.mockito.Mockito.*;
import java.util.ArrayList;
import java.util.List;
import org.junit.Test;
import org.mockito.Mockito;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import tacos.Ingredient.Type;
import tacos.data.TacoRepository;
import tacos.web.api.DesignTacoController;

public class DesignTacoControllerTest {

  @Test
  public void shouldReturnRecentTacos() {
    Taco[] tacos = {
        testTaco(1L), testTaco(2L),
        testTaco(3L), testTaco(4L),           ←--- 创建测试数据
        testTaco(5L), testTaco(6L),
        testTaco(7L), testTaco(8L),
        testTaco(9L), testTaco(10L),
        testTaco(11L), testTaco(12L),
        testTaco(13L), testTaco(14L),
        testTaco(15L), testTaco(16L)};
    Flux<Taco> tacoFlux = Flux.just(tacos);

    TacoRepository tacoRepo = Mockito.mock(TacoRepository.class);
    when(tacoRepo.findAll()).thenReturn(tacoFlux);        ←--- Mock Taco
Repository

    WebTestClient testClient = WebTestClient.bindToController(
        new DesignTacoController(tacoRepo))
          .build();                               ←--- 创建WebTestClient
```

```
    testClient.get().uri("/design/recent")
      .exchange()                              ←--- 请求最近的taco
      .expectStatus().isOk()                ←--- 检验预期的响应
      .expectBody()
        .jsonPath("$").isArray()
        .jsonPath("$").isNotEmpty()
        .jsonPath("$[0].id").isEqualTo(tacos[0].getId().toString())
        .jsonPath("$[0].name").isEqualTo("Taco 1").jsonPath("$[1].id")
        .isEqualTo(tacos[1].getId().toString()).jsonPath("$[1].name")
        .isEqualTo("Taco 2").jsonPath("$[11].id")
        .isEqualTo(tacos[11].getId().toString())

  ...
        .jsonPath("$[11].name").isEqualTo("Taco 12").jsonPath("$[12]")
     .doesNotExist();
        .jsonPath("$[12]").doesNotExist();
  }

  ...

}
```

shouldReturnRecentTacos()方法做的第一件事情就是以Flux<Taco>的形式创建了一些测试数据。这个Flux随后作为mock TacoRepository的findAll()方法的返回值。

Flux发布的Taco对象是由一个名为testTaco()的方法创建的。这个方法会根据一个数字生成一个Taco，其ID和名称都是基于该数字生成的。testTaco()方法的实现如下所示：

```
private Taco testTaco(Long number) {
    Taco taco = new Taco();
    taco.setId(UUID.randomUUID());
    taco.setName("Taco " + number);
    List<IngredientUDT> ingredients = new ArrayList<>();
    ingredients.add(
        new IngredientUDT("INGA", "Ingredient A", Type.WRAP));
    ingredients.add(
        new IngredientUDT("INGB", "Ingredient B", Type.PROTEIN));
    taco.setIngredients(ingredients);
    return taco;
```

```
}
```

简单起见，所有的测试taco都具有两种相同的配料，但是它们的ID和名称是根据传入的数字确定的。

另外，回到shouldReturnRecentTacos()方法，我们实例化了一个DesignTacoController并将mock TacoRepository注入到了构造器中。这个控制器传递给了WebTestClient. bindToController()方法，以便于生成WebTestClient实例。

所有的环境搭建工作完成后，我们可以使用WebTestClient提交GET请求至“/design/recent”并校验响应符合我们的预期。对get().uri("/design/recent")的调用描述了我们想要发送的请求。随后，调用exchange()会提交请求，这个请求将会由WebTestClient绑定的控制器（DesignTacoController）来进行处理。

最后，我们可以确认响应符合预期。通过调用expectStatus()，我们可以断言响应具有HTTP 200 (OK)状态码。然后，我们多次调用jsonPath()断言响应体中的JSON包含它应该具有的值。最后一个断言检查第12个元素（在基于零开始计数的数组中）是否真的不存在，以此判断结果不超过12个元素。

如果返回的JSON比较复杂，比如有大量的数据或多层嵌套的数据，那么使用jsonPath()会变得非常烦琐。实际上，为了节省空间，在程序清单11.1中，我省略了很多对jsonPath()的调用。在这种情况下，使用jsonPath()会变得非常枯燥烦琐，WebTestClient提供了json()方法。这个

方法可以传入一个String参数（包含响应要对比的JSON）。

举例来说，假设我们在名为recent-tacos.json的文件中创建了完整的响应JSON并将它放到了类路径的“/tacos”路径下，那么我们可以按照如下的方式重写WebTestClient断言：

```
ClassPathResource recentsResource =
    new ClassPathResource("/tacos/recent-tacos.json");
String recentsJson = StreamUtils.copyToString(
    recentsResource.getInputStream(), Charset.defaultCharset());

testClient.get().uri("/design/recent")
  .accept(MediaType.APPLICATION_JSON)
  .exchange()
  .expectStatus().isOk()
  .expectBody()
    .json(recentsJson);
```

因为json()接受的是一个String，所以我们必须先将类路径资源加载为String。借助Spring中StreamUtils的copyToString()方法，这一点很容易实现。copyToString()方法返回的String就是我们的请求所预期的响应JSON内容。将其传递给json()方法，我们就能确保控制器会生成正确的输出。

WebTestClient提供的另外一种可选方案就是它允许将响应体与一个值的列表进行对比。expectBodyList()方法会接受一个代表列表中元素类型的Class或ParameterizedTypeReference，并且会返回ListBodySpec对象，随后可以基于该对象进行断言。借助expectBodyList()，我们可以重写测试类，使用创建mock TacoRepository时的测试数据的子集来进行验证：

```
testClient.get().uri("/design/recent")
  .accept(MediaType.APPLICATION_JSON)
  .exchange()
  .expectStatus().isOk()
  .expectBodyList(Taco.class)
    .contains(Arrays.copyOf(tacos, 12));
```

在这里，我们断言响应体包含了在测试方法开头处所创建的原始Taco数组的前12个元素。

11.3.2　测试POST请求

WebTestClient不仅能对控制器发送GET请求，还能用来测试各种HTTP方法，包括GET、POST、PUT、PATCH、DELETE和HEAD方法。表11.1将HTTP方法与WebTestClient的方法进行了映射。

表**11.1**　**WebTestClient**能够测试针对**Spring WebFlux**控制器的各种请求

HTTP方法	**WebTestClient**方法
GET	.get()
POST	.post()
PUT	.put()
PATCH	.patch()

DELETE	.delete()
HEAD	.head()

作为测试Spring WebFlux控制器其他HTTP请求方法的样例，我们看一下针对DesignTacoController的另一个测试。这一次，我们会编写一个对taco创建API的测试，也就是提交POST请求到“/design”：

```
@Test
public void shouldSaveATaco() {
  TacoRepository tacoRepo = Mockito.mock(
              TacoRepository.class);             ←--- 搭建测试数据
  Mono<Taco> unsavedTacoMono = Mono.just(testTaco(null));
  Taco savedTaco = testTaco(null);
  savedTaco.setId(1L);
  Mono<Taco> savedTacoMono = Mono.just(savedTaco);
  when(tacoRepo.save(any())).thenReturn(savedTacoMono);     ←--- mock Tac
oRepository

  WebTestClient testClient = WebTestClient.bindToController(    ←--- 创建W
ebTestClient
      new DesignTacoController(tacoRepo)).build();

  testClient.post()                 ←--- POST taco
      .uri("/design")
      .contentType(MediaType.APPLICATION_JSON)
      .body(unsavedTacoMono, Taco.class)
    .exchange()
    .expectStatus().isCreated()        ←--- 校验响应
    .expectBody(Taco.class)
      .isEqualTo(savedTaco);
}
```

与上面的测试方法类似，shouldSaveATaco()首先会创建一些测试数据和mock TacoRepository，并且创建了一个WebTestClient并绑定到控制器上。随后，它使用WebTestClient提交POST请求到“/design”，并且将

请求体声明为application/json类型，请求载荷为Taco的JSON序列化形式，放到未保存的Mono中。在执行exchange()之后，测试断言响应状态为HTTP 201 (CREATED)并且响应体中的载荷与已保存的Taco对象相同。

11.3.3 使用实时服务器进行测试

到目前为止，我们所编写的测试都依赖于Spring WebFlux的mock实现，所以并不需要真正的服务器。但是，我们可能需要在服务器（如Netty或Tomcat）环境中测试WebFlux控制器，也许还会需要repository或其他的依赖。换句话说，我们有可能要编写集成测试。

要编写WebTestClient的集成测试，与其他的Spring Boot集成测试类似，我们首先要为测试类添加@RunWith和@SpringBootTest：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class DesignTacoControllerWebTest {

  @Autowired
  private WebTestClient testClient;

}
```

通过将webEnvironment属性设置为WebEnvironment.RANDOM_PORT，我们要求Spring启动一个运行时服务器并监听任意选择的端口[1]。

你可能也注意到，我们将WebTestClient自动织入到了测试类中。这

不仅意味着我们不用在测试的方法中创建它了，而且在发送请求的时候也不需要指定完整的URL了。这是因为WebTestClient能够知道测试服务器在哪个端口上运行。现在，我们可以使用自动织入的WebTestClient将shouldReturnRecentTacos()重写为集成测试：

```
@Test
public void shouldReturnRecentTacos() throws IOException {
  testClient.get().uri("/design/recent")
    .accept(MediaType.APPLICATION_JSON).exchange()
    .expectStatus().isOk()
    .expectBody()
        .jsonPath("$[?(@.id == 'TACO1')].name")
            .isEqualTo("Carnivore")
        .jsonPath("$[?(@.id == 'TACO2')].name")
            .isEqualTo("Bovine Bounty")
        .jsonPath("$[?(@.id == 'TACO3')].name")
            .isEqualTo("Veg-Out");
}
```

我们发现，这个新版本的shouldReturnRecentTacos()代码要少得多。我们不再需要创建WebTestClient，因为可以使用自动织入的实例。另外，也不需要mock TacoRepository，因为Spring将会创建DesignTacoController实例并将一个真正的TacoRepository注入进来。在新版本的测试方法中，我们使用JSONPath表达式来校验数据库提供的值。

WebTestClient在测试的时候非常有用，此时我们会消费WebFlux控制器所暴露的API。但是，如果我们的应用本身要消费某个API，又该怎样处理呢？接下来，我们将注意力转向Spring反应式Web的客户端，看一下WebClient如何通过REST客户端来处理反应式类型，如Mono和Flux。

11.4　反应式消费REST API

在第7章中，我们使用RestTemplate发送客户端请求到Taco Cloud API上。RestTemplate有着很久的历史，从Spring 3.0版本就引入了。我们曾经使用它为应用发送了无数的请求，但是RestTemplate提供的方法处理的都是非反应式领域类型和集合。这意味着，如果我们想要以反应式的方式使用响应数据，就需要使用Flux或Mono对其进行包装。如果我们已经有了Flux或Mono，想要通过POST或PUT请求发送它们，那么我们需要在发送请求之前将数据抽取到一个非反应式的类型中。

如果能够有一种方式让RestTemplate原生使用反应式类型那就好了。不用担心，Spring 5提供了WebClient，它可以作为RestTemplate的反应式版本。WebClient能够让我们请求外部API时发送和接收反应式类型。

WebClient的使用方式与RestTemplate有很大的差别。RestTemplate会有多个方法处理不同类型的请求；而WebClient有一个流畅（fluent）的构建者风格接口，能够让我们描述和发送请求。WebClient的通用使用模式如下：

- 创建WebClient实例（或注入WebClient bean）；
- 指定要发送请求的HTTP方法；
- 指定请求中URI和头信息；
- 提交请求；
- 消费响应。

接下来，我们实际看几个WebClient的例子，首先从如何使用WebClient发送HTTP GET请求开始。

11.4.1 获取资源

作为使用WebClient的样例，假设我们需要通过Taco Cloud API根据ID获取Ingredient对象。如果使用RestTemplate，那么我们可能会使用getForObject()方法。但是，借助WebClient的话，我们会构建请求、获取响应并抽取一个会发布Ingredient对象的Mono：

```
Mono<Ingredient> ingredient = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);

ingredient.subscribe(i -> { ... })
```

在这里，我们使用create()创建了一个新的WebClient实例。然后，我们使用get()和uri()定义对http://localhost:8080/ingredients/{id}的GET请求，其中{id}占位符将会被ingredientId的值所替换。接着，retrieve()会执行请求。最后，我们调用bodyToMono()将响应体的载荷抽取到Mono<Ingredient>中，就可以继续使用Mono的额外操作了。

为了对bodyToMono()返回Mono进行额外的操作，需要注意的很重要的一点是要在请求发送之前对其进行订阅。发送请求获取值的集合是非常容易的。例如，如下的代码片段将获取所有配料：

```
Flux<Ingredient> ingredients = WebClient.create()
```

```
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients.subscribe(i -> { ... })
```

大部分而言，获取多个条目与获取单个条目是相同的。最大的差异在于我们不再是使用bodyToMono()将响应体抽取为Mono，而是使用bodyToFlux()将其抽取为一个Flux。

与bodyToMono()类似，bodyToFlux()返回的Flux还没有被订阅。在数据流过之前，我们可以对Flux添加一些额外的操作（过滤、映射等）。因此，非常重要的一点就是要订阅结果所形成的Flux，否则请求将始终不会发送。

使用基础URI发送请求

你可能会发现在很多请求中都会使用一个通用的基础URI。这样的话，创建WebClient bean的时候设置一个基础URI并将其注入到所需的地方是非常有用的。这样的bean可以按照如下的方式来声明：

```
@Bean
public WebClient webClient() {
  return WebClient.create("http://localhost:8080");
}
```

然后，在想要使用基础URI的任意地方，我们都可以将WebClient bean注入进来并按照如下的方式来使用：

```
@Autowired
WebClient webClient;
```

```
public Mono<Ingredient> getIngredientById(String ingredientId) {
  Mono<Ingredient> ingredient = webClient
    .get()
    .uri("/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);

  ingredient.subscribe(i -> { ... })
}
```

因为WebClient已经创建好了，所以我们可以通过get()方法直接使用它。对于URI来说，我们只需要调用uri()指定相对于基础URI的相对路径即可。

对长时间运行的请求进行超时处理

我们需要考虑的一件事情就是，网络并不是始终可靠的，或者并不像我们预期的那么快，远程服务器在处理请求时有可能会非常缓慢。理想情况下，对远程服务的请求会在一个合理的时间内返回。无法正常返回的话，客户端要是能够避免陷入长时间等待响应的窘境就好了。

为了避免客户端请求被缓慢的网络或服务阻塞，我们可以使用Flux或Mono的timeout()方法，为等待数据发布的过程设置一个时长限制。作为样例，我们考虑一下如何为获取配料数据使用timeout()方法：

```
Flux<Ingredient> ingredients = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients
  .timeout(Duration.ofSeconds(1))
  .subscribe(
      i -> { ... },
```

```
      e -> {
        // handle timeout error
      })
```

可以看到，在订阅Flux之前，我们调用了timeout()方法，将持续时间设置成了1秒。如果请求能够在1秒之内返回，就不会有任何问题。如果请求的耗时超过1秒，就会超时，作为第二个参数传递给subscribe()的错误处理器将会被调用。

11.4.2　发送资源

使用WebClient发送数据与接收数据并没有太大的差异。作为样例，假设我们有一个Mono<Ingredient>，并且想要将Mono发布的Ingredient对象以POST请求的形式发送到相对路径“/ingredients”的URI上。我们所需要做的就是使用post()方法来替换get()，并通过body()方法指明要使用Mono来填充请求体：

```
Mono<Ingredient> ingredientMono = ...;

Mono<Ingredient> result = webClient
  .post()
  .uri("/ingredients")
  .body(ingredientMono, Ingredient.class)
  .retrieve()
  .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })
```

如果我们没有要发送的Mono或Flux，而只有原始的领域对象，那么可以使用syncBody()方法。例如，假设我们没有Mono<Ingredient>，而是有一个想要在请求体中发送的Ingredient对象，那么可以这样做：

```
Ingedient ingredient = ...;

Mono<Ingredient> result = webClient
  .post()
  .uri("/ingredients")
  .syncBody(ingredient)
  .retrieve()
  .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })
```

如果我们不是使用POST请求，而是想要使用PUT请求更新一个Ingredient，就可以使用put()来替换post()，并相应地调整URI路径：

```
Mono<Void> result = webClient
  .put()
  .uri("/ingredients/{id}", ingredient.getId())
  .syncBody(ingredient)
  .retrieve()
  .bodyToMono(Void.class)
  .subscribe();
```

PUT请求的响应载荷一般是空的，所以我们必须要求bodyToMono()返回一个Void类型的Mono。一旦订阅该Mono，请求就会立即发送。

11.4.3 删除资源

WebClient还支持通过其delete()方法移除资源。例如，根据ID删除配料：

```
Mono<Void> result = webClient
  .delete()
  .uri("/ingredients/{id}", ingredientId)
  .retrieve()
  .bodyToMono(Void.class)
  .subscribe();
```

与PUT请求类似，DELETE请求的响应不会有载荷。同样，我们返回并订阅Mono<Void>就会发送请求。

11.4.4　处理错误

到目前为止，所有的WebClient样例都假设有一个正常的结果：没有400级别和500级别的状态码。如果出现这两种类型的错误状态，WebClient就会记录失败信息；否则，就会默默忽略掉。

如果你需要处理这种错误，那么可以调用onStatus()来指定各种类型的HTTP状态码该如何进行处理。onStatus()接受两个函数：一个断言函数用来匹配HTTP状态；另一个函数会得到ClientResponse对象，并返回Mono<Throwable>。

为了阐述如何使用onStatus()创建自定义的错误处理器，请参考如下使用WebClient根据ID获取配料的样例：

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

如果ingredientId的值能够匹配已知的资源，那么结果得到的Mono在订阅时就会发布一个Ingredient。但是，如果找不到匹配的配料呢？

当订阅可能会出现错误的Mono或Flux时，很重要的一点就是在调用subscribe()注册数据消费者的同时，也要注册一个错误消费者：

```
ingredientMono.subscribe(
    ingredient -> {
      // handle the ingredient data
      ...
    },
    error-> {
      // deal with the error
      ...
    });
```

如果能够找到配料资源，那么传递给subscribe()的第一个lambda表达式（数据消费者）将会被调用，并且会将匹配的Ingredient对象传递过来。但是，如果找不到资源，那么请求将会得到一个HTTP 404 (NOT FOUND)状态码的响应，它将会导致第二个lambda表达式（错误消费者）被调用，并且会传递过来一个默认的WebClientResponseException。

WebClientResponseException最大的问题在于它无法明确指出导致Mono失败的原因是什么。它的名字表明在WebClient发起的请求中，出现了响应错误，但是我们需要深入研究WebClientResponseException才能知道哪里出现了错误。无论如何，如果给错误消费者的异常更加专注业务领域而不是专注WebClient，那就更好了。

我们可以添加一个自定义的错误处理器，在这个处理器中可以提供将状态码转换为自己所选择的Throwable的代码。如果请求配料资源时得到的Mono失败，我们就生成一个UnknownIngredientException。在调用retrieve()之后，我们可以添加一个对onStatus()的调用，从而实现这一点：

```
Mono<Ingredient> ingredientMono = webClient
```

```
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(HttpStatus::is4xxClientError,
            response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

调用onStatus()时第一个参数是断言，它会接受一个HttpStatus，如果状态码是我们想要处理的，就将会返回true。如果状态码匹配，响应将会传递给第二个参数的函数并按需进行处理，最终返回Throwable类型的Mono。

在样例中，如果状态码是400级别的（比如客户端错误），那么将会返回包含UnknownIngredientException的Mono。这会导致ingredientMono因为该异常而失败。

需要注意，HttpStatus::is4xxClientError是对HttpStatus的is4xxClientError的方法引用。此时，将会基于HttpStatus对象调用该方法。如果喜欢，还可以使用HttpStatus的其他方法作为方法引用。你也可以以lambda表达式或方法引用的形式提供其他返回boolean类型的函数。

例如，在错误处理中，我们可以更加精确地检查HTTP 404 (NOT FOUND)状态，只需将对onStatus()的调用修改成如下形式即可：

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(status -> status == HttpStatus.NOT_FOUND,
            response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

值得一提的是，我们可以按需调用onStatus()任意多次，以便于处理响应中可能返回的各种HTTP状态码。

11.4.5 交换请求

到目前为止，在使用WebClient的时候，我们都是利用它的retrieve()方法来发送请求。在这些场景中，retrieve()方法会返回一个ResponseSpec类型的对象，通过调用它的onStatus()、bodyToFlux()和bodyToMono()方法，我们就能处理响应。对于简单的场景来说，使用ResponseSpec就足够了，但是它在很多方面都有局限性。如果我们想要访问响应的头信息或cookie的值，那么ResponseSpec就无能为力了。

在使用ResponseSpec遇到困难时，我们就可以通过调用exchange()方法来替换retrieve()方法。exchange()方法会返回ClientResponse类型的Mono，我们可以对它采用各种反应式操作，以便于探测和使用整个响应中的数据，包括载荷、头信息和cookie。

在了解exchange()和retrieve()的差异之前，我们先看一下它们之间的相似之处。如下的代码片段通过WebClient和exchange()方法，根据ID获取单个配料：

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchange()
    .flatMap(cr -> cr.bodyToMono(Ingredient.class));
```

这几乎与使用retrieve()的样例是相同的：

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

在exchange()样例中，我们不是使用ResponseSpec对象的bodyToMono()方法来获取Mono<Ingredient>，而是得到了一个Mono<ClientResponse>，通过它我们可以执行扁平化映射（flat-mapping）函数，将ClientResponse映射为Mono<Ingredient>，这样扁平化为最终想要的Mono。

现在，我们看一下exchange()的差异在什么地方。假设请求的响应中会包含一个名为X_UNAVAILABLE的头信息，如果它的值为true，则表明该配料是不可用的（因为某种原因）。为了讨论方便，假设如果这个头信息存在，那么我们希望得到的Mono是空的，不返回任何内容。通过添加另外一个flatMap()调用，我们就能实现这一点。整个的WebClient调用过程如下所示：

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchange()
    .flatMap(cr -> {
      if (cr.headers().header("X_UNAVAILABLE").contains("true")) {
        return Mono.empty();
      }
      return Mono.just(cr);
    })
    .flatMap(cr -> cr.bodyToMono(Ingredient.class));
```

新的flatMap()调用会探查给定ClientRequest对象的响应头，查看是否存在值为true的X_UNAVAILABLE头信息。如果能够找到，就将会返

回一个空的Mono；否则，返回一个包含ClientResponse的新Mono。不管是哪种情况，返回的Mono都会扁平化为下一个flatMap()操作所要使用的Mono。

11.5　保护反应式Web API

从Spring Security诞生以来（甚至可以追溯到它叫作Acegi Security的时代），它的Web安全模型就是基于Servlet Filter构建的。毕竟，这样做是有道理的。如果我们希望拦截基于Servlet技术的Web框架的请求，以确保该请求得到了恰当的授权，那么Servlet Filter是显而易见的方案。但是，Spring WebFlux并不适用于这种方式。

在使用Spring WebFlux编写Web应用的时候，我们甚至都不能保证会用到Servlet。实际上，反应式Web应用很有可能构建在Netty或其他非Servlet容器上。这是否意味着基于Servlet Filter的Spring Security不能用来保护我们的Spring WebFlux应用了呢？

在保护Spring WebFlux应用的时候，Servlet Filter确实不是可行方案了。但是，Spring Security依然可以胜任这项任务。从5.0.0版本开始，Spring Security就既能保护基于Servlet的Spring MVC，又能保护反应式的Spring WebFlux应用了。在实现这一点的时候，它使用了Spring的WebFilter，这是Spring模仿Servlet Filter的类似方案，但是它不依赖于Servlet API。

然而，更值得注意的是，反应式Spring Security的配置模型与在第4

章中看到的没有太大不同。事实上，Spring WebFlux与Spring MVC有着独立的依赖关系，但与之不同，Spring Security是作为同一个Spring Boot Security starter提供的，不管你是打算使用它来保护Spring MVC Web应用，还是保护使用Spring WebFlux编写的应用，都需要添加这项依赖。提醒一下，security starter如下所示：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

也就是说，Spring Security的反应式和非反应式配置模型仅有几项很小的差异。我们很有必要快速对比一下这两种配置模型。

11.5.1 配置反应式Web应用的安全性

回忆一下，配置Spring Security来保护Spring MVC Web应用通常需要创建一个扩展自WebSecurityConfigurerAdapter的新配置类，并使用@EnableWebSecurity注解。这样的配置类将重写configuration()方法，以指定Web安全的细节，例如特定的请求路径需要哪些权限。下面这个简单的Spring Security配置类可以帮助我们回忆如何为非反应式Spring MVC应用配置安全性：

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
```

```
      .authorizeRequests()
        .antMatchers("/design", "/orders").hasAuthority("USER")
        .antMatchers("/**").permitAll();
  }

}
```

现在，我们看一下相同的配置如何用到反应式Spring WebFlux应用中。程序清单11.2展现了一个反应式安全配置类，它的功能与前文的安全配置大致相同：

程序清单11.2　为Spring WebFlux配置Spring Security

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

  @Bean
  public SecurityWebFilterChain securityWebFilterChain(
                                             ServerHttpSecurity http) {
    return http
        .authorizeExchange()
          .pathMatchers("/design", "/orders").hasAuthority("USER")
          .anyExchange().permitAll()
      .and()
        .build();
  }

}
```

我们可以看到，有很多类似的地方，同时也有所差异。这个新的配置类没有使用@EnableWebSecurity，而是使用了@EnableWebFluxSecurity注解。除此之外，配置类没有扩展WebSecurityConfigurerAdapter或其他的基类，因此也就没有必要重写configure()。

为了取代configure()的功能，我们通过securityWebFilterChain()方法

声明了一个SecurityWebFilterChain类型的bean。securityWebFilterChain()的方法体与前面配置的configure()方法没有太大的差异，但是也有略微的修改。

最重要的是，配置是通过给定的ServerHttpSecurity对象进行声明的，而不是通过HttpSecurity对象。借助ServerHttpSecurity，我们可以调用authorizeExchange()，它大致等价于authorizeRequests()，都是用来声明请求级的安全性的。

> 注意：ServerHttpSecurity是Spring Security 5新引入的，在反应式编程中它模拟了HttpSecurity的功能。

在映射路径的时候，我们依然可以使用Ant风格的通配符路径，但是这里要使用pathMatchers()，而不是antMatchers()。这样做的结果就是，我们不再需要声明Ant风格的路径“/**”来捕获所有请求，因为anyExchange()会映射所有的路径。

最后，因为我们将SecurityWebFilterChain声明为一个bean，而不是重写框架方法，所以我们需要调用build()方法将所有的安全规则聚合到一个要返回的SecurityWebFilterChain对象中。

除了这些微小的差异外，配置Spring WebFlux和Spring MVC的Web安全性并没有太多不同。那么如何获取用户的详情信息呢？

11.5.2 配置反应式的用户详情服务

在扩展WebSecurityConfigurerAdapter的时候，我们会重写configure()方法以声明安全规则，并且还会重写另外一个configure()方法来配置认证逻辑，通常这需要定义一个UserDetails。为了提醒一下代码会是什么样子，如下的代码重写了configure()方法，并且在UserDetailsService的匿名实现中，使用了注入的UserRepository对象以提供根据用户名查找用户的功能：

```
@Autowired
UserRepository userRepo;

@Override
protected void
    configure(AuthenticationManagerBuilder auth)
    throws Exception {
  auth
    .userDetailsService(new UserDetailsService() {
      @Override
      public UserDetails loadUserByUsername(String username)
                                   throws UsernameNotFoundException {
        User user = userRepo.findByUsername(username)
        if (user == null) {
          throw new UsernameNotFoundException(
                         username " + not found")
        }
        return user.toUserDetails();
      }
    });
}
```

在这个非反应式的配置中，我们重写了UserDetailsService唯一要求的方法，也就是loadUserByUsername()。在这个方法内部，我们使用给定的UserRepository，实现根据用户名来查找用户的功能。如果没有找

到该名称的用户，就会抛出UsernameNotFoundException。如果能够找到，就调用一个辅助方法toUserDetails()，返回最终的UserDetails对象。

在反应式的安全配置中，我们不再重写configure()方法，而是声明一个ReactiveUserDetailsService bean。ReactiveUserDetailsService是UserDetailsService的反应式等价形式。与UserDetailsService类似，ReactiveUserDetailsService只需要实现一个方法。具体来讲，就是一个返回Mono<UserDetails>的findByUsername()方法，这里返回的不再是UserDetails对象。

在下面的样例中，ReactiveUserDetailsService bean会给定一个UserRepository，我们假设它是一个反应式的Spring Data repository（在第12章我们将会详细讨论）：

```
@Service
public ReactiveUserDetailsService userDetailsService(
                                          UserRepository userRepo) {
  return new ReactiveUserDetailsService() {
    @Override
    public Mono<UserDetails> findByUsername(String username) {
      return userRepo.findByUsername(username)
        .map(user -> {
          return user.toUserDetails();
        });
    }
  };
}
```

在这里，按需要返回一个Mono<UserDetails>，但是UserRepository.findByUsername()方法所返回的是Mono<User>。因为它是一个Mono，所以可以对它进行链式操作，比如进行map()操作，将

Mono<User>映射为Mono<UserDetails>。

在本例中，map()操作使用了一个lambda表达式，它调用了Mono所发布的User对象上的toUserDetails()方法。这个方法会将User转换为UserDetails。这样的话，“.map()”操作会返回一个Mono<UserDetails>，恰好就是ReactiveUserDetailsService.findByUsername()方法所需要的。

11.6 小结

- Spring WebFlux提供了一个反应式的Web框架，它的编程模型是与Spring MVC对应的，甚至共享了很多相同的注解。
- Spring 5还提供了函数式编程模型，作为Spring WebFlux的替代方案。
- 反应式控制器可以使用WebTestClient来进行测试。
- 在客户端，Spring 5提供了WebClient，也就是Spring RestTemplate的反应式等价实现。
- 在保护Web应用方面，尽管WebFlux在底层有一些区别，但是Spring Security 5为反应式安全所提供的编程模型与非反应式Spring MVC应用相比并没有特别大的差异。

---

[1] 我们也可以将webEnvironment设置为WebEnvironment.DEFINED_PORT，并利用配置属性指定一个端口，但是强烈建议不要这样做。这样做的话将会带来并行运行服务器端口冲突的风险。

第12章　反应式持久化数据

本章内容：

- Spring Data的反应式repository
- 为Cassandra和MongoDB编写反应式repository
- 以反应式的方式使用非反应式的repository
- Cassandra的数据模型

在思考非阻塞的反应式代码和阻塞的命令式代码时，我经常会想到上下班的高峰时刻（rush hour）。高峰时刻是一个很奇怪的名字。每个人都急着去他们想到的地方，但是我们通常只能几乎一动不动地坐在车流之中。如果路上没有其他人，我们能够轻而易举地到达目的地。

即便我非常希望到达某个地方（我没有阻塞），但是这并不意味着路上没有其他人挡着我。前面可能有其他司机发生了剐蹭事故，阻塞了其他车辆的通行。所以即使我本可以畅通无阻地回到家中，但此刻我也只能阻塞在这里等待事故清理完成。

在前面的章节中，我们看到了如何使用Spring WebFlux创建反应式、非阻塞的控制器。这样能够帮助我们提升Web层的可扩展性。但是，只有当与这些控制器协作的其他组件都是非阻塞的时候，它们本身才能是非阻塞的。如果我们编写的Spring WebFlux控制器依赖于阻塞的repository，那么反应式控制器需要阻塞等待它们生成数据。

因此，很重要的一点在于，要让整个数据流变成反应式和非阻塞的，也就是从控制器直到数据库。在本章中，我们将会看到如何使用Spring Data编写反应式的repository，这些repository与我们在第3章看到的编程模型非常类似。我们首先从整体上了解一下Spring Data对反应式的支持。

12.1 理解Spring Data的反应式概况

从Spring Data Kay release train开始，Spring Data首次提供对反应式repository的支持，其中包括使用Cassandra、MongoDB、Couchbase或Redis持久化数据的反应式编程模型。

名称的由来

尽管Spring Data的各个项目都有自己的节奏，但是它们都按照一个release train来进行发布，每个release train的命名对应计算机科学中一个重要人物的名字。

这些名字是按照字母排序的，比如Babbage、Codd、

Dijkstra、Evans、Fowler、Gosling、Hopper和Ingalls。在编写本书的时候，最新的release train版本是Spring Data Kay，这是根据Alan Kay来命名的，Alan Kay是Smalltalk编程语言的设计者之一。

你可能也发现了，在这里我并没有提到关系型数据库或JPA。令人遗憾的是，目前还没有对反应式JPA的支持。尽管关系型数据库依然是行业中使用最广泛的数据库方案，但是要让Spring Data JPA支持反应式编程模型，需要数据库和相关的JDBC都支持非阻塞的反应式模型。不幸的是，至少目前还不支持关系数据库的反应式处理。希望这种情况能在不久的将来得到解决。[1]

本章的重点是使用Spring Data为支持反应式模型的数据库开发使用反应式类型的repository。我们首先对比一下Spring Data的反应式模型和非反应式模型。

12.1.1 Spring Data反应式本质论

Spring Data反应式的本质可以概括为一句话，那就是在反应式repository的方法中，要接受和返回Mono和Flux，而不是领域实体和集合。根据配料类型，从后端数据库中获取Ingredient对象的repository，可以声明为如下的repository接口：

```
Flux<Ingredient> findByType(Ingredient.Type type);
```

我们可以看到，这个findByType()方法会返回Flux<Ingredient>，而不是像对应的非反应式实现那样返回List<Ingredient>或Iterable<Ingredient>。

类似的，在保存Taco的时候，repository的saveAll()方法签名如下所示：

```
<Taco> Flux<Taco> saveAll(Publisher<Taco> tacoPublisher);
```

在本例中，saveAll()方法接受一个Taco类型的Publisher（可能是Mono<Taco>或Flux<Taco>）并返回一个Flux<Taco>。这与非反应式的repository是不同的，它的save()方法直接处理领域类型，接受Taco对象并返回保存的Taco对象。

简而言之，Spring Data的反应式repository与我们在第3章看到的Spring Data的非反应式repository共享几乎相同的编程模型。唯一重要的区别是，反应式repository的方法接受和返回Flux和Mono，而不是原始的领域类型和集合。

12.1.2 反应式和非反应式类型之间的转换

在进一步研究如何使用Spring Data编写反应式repository之前，我们看一下如何解决遗留的巨大问题。我们可能已经使用了关系型数据库，将数据迁移至Spring Data反应式编程模型支持的4种数据库之一是不太现实的，那是否就意味着我们无法在应用中使用反应式编程了呢？

从头到尾使用反应式模型（包括数据库层面）时，我们才能够得到反应式编程的全部收益，但是在非反应式数据库之上使用反应式流的话，我们也能得到一部分收益。即便我们所选择的数据库不支持非阻塞的反应式查询，我们依然可以以阻塞的方式获取数据并将其转换为反应式类型，从而使上游组件从中收益。

例如，假设我们正在使用关系型数据库并利用Spring Data JPA进行持久化。我们的OrderRepository可能会有一个如下签名的方法：

```
List<Order> findByUser(User user);
```

这个方法会返回一个非反应式的List<Order>，包含给定User的所有Order信息。当findByUser()被调用的时候，查询执行的过程中该方法会阻塞，结果会收集到一个List中。因为List并不是反应式类型，所以我们不能在它上面执行Flux提供的任何操作。另外，如果调用者是控制器，那么它无法以反应式的方式处理结果，实现提高可扩展性的目的。

在JPA repository的阻塞性方面我们确实无能为力。但是，我们可以在接收到非反应式List的时候就将其转换成Flux，这样我们就可以从这里开始以反应式的方式处理结果了。为了实现这一点，我们可以使用Flux.fromIterable()：

```
List<Order> orders = repo.findByUser(someUser);
Flux<Order> orderFlux = Flux.fromIterable(orders);
```

与之类似，如果我们想要根据ID获取一个Order，我们就可以立即将其转换为Mono：

```
Order order repo.findById(Long id);
Mono<Order> orderMono = Mono.just(order);
```

通过使用Mono.just()和Flux的fromIterable()、fromArray()和fromStream()方法，我们可以将非反应式阻塞代码隔离在repository中，在应用的其他地方，我们都可以使用反应式类型。

那反方向怎么样呢？如果我们有一个Mono或Flux，此时需要调用非反应式JPA repository的save()方法又该怎么办呢？好消息是，Mono和Flux都提供了将它们发布的数据抽取到领域类型或Iterable中的操作。

例如，假设WebFlux控制器接受的是Mono<Taco>，那么我们需要使用Spring Data JPA repository的save()方法将其保存起来。没有问题，我们只需调用Mono的block()方法就可以抽取Taco对象：

```
Taco taco = tacoMono.block();
tacoRepo.save(taco);
```

顾名思义，block()方法会执行一个阻塞操作，完成数据的抽取过程。

如果要从Flux中抽取数据，我们可以使用toIterable()。假设我们有一个Flux<Taco>，并且要调用Spring Data JPA repository的saveAll()方法，如下的代码片段将从Flux<Taco>中抽取Iterable<Taco>：

```
Iterable<Taco> tacos = tacoFlux.toIterable();
tacoRepo.saveAll(tacos);
```

与Mono.block()类似，Flux.toIterable()在将Flux发布的对象抽取到

Iterable的过程中是阻塞的。因为它们本质上是阻塞的，所以应该谨慎使用Mono.block()和Flux.toIterable()，并且要清楚地认识到使用它们会打破反应式编程模型。

要避免阻塞的抽取操作，还有一种更具反应式的方法，就是订阅Mono或Flux，并在其发布每个元素的时候执行所需的操作。例如，要使用非反应式的repository保存Flux<Taco>发布的Taco对象，我们可以这样做：

```
tacoFlux.subscribe(taco -> {
  tacoRepo.save(taco);
});
```

虽然调用repository的save()方法依然是非反应式的阻塞操作，但是在消费和处理Flux或Mono发布的数据时，使用subscribe()是一种更自然、更加反应式的方式。

关于非反应式repository，我们已经讨论得够多了。接下来，我们见识一下Spring Data反应式功能的真正威力，为Taco Cloud应用创建反应式repository。

12.1.3 开发反应式repository

正如我们在第3章中看到的那样，Spring Data最令人赞叹的特性之一就是我们只须声明repository接口即可，在运行时Spring Data会自动实现它们。在那一章中，我们主要关注Spring Data JPA，但是同样的编程模型也适用于非关系数据库，包括Cassandra和MongoDB。

除了Spring Data Cassandra和Spring Data MongoDB对非反应式repository的支持之外，它们都提供了反应式的编程模型。这些数据库在后端提供数据持久化功能，Spring应用可以真正实现从Web层到数据库的端到端反应式流。我们首先看看如何使用反应式Spring Data repository将数据持久化到Cassandra。

12.2 使用反应式的Cassandra repository

Cassandra是一个分布式、高性能、始终可用、最终一致、分区行存储的NoSQL数据库。

描述该数据库的形容词是非常冗长的，但每一个词都准确说明了Cassandra的威力。简而言之，Cassandra处理的是数据行（row of data），这些数据行会在多个分布式节点中分区。不会有任何节点保存所有的数据，但是任何给定的行都会跨多个节点保存副本，从而消除了单点故障。

Spring Data Cassandra为Cassandra数据库提供了自动化repository的支持，这与Spring Data JPA为关系数据库提供的支持非常相似，但又有着明显的差异。此外，Spring Data Cassandra还提供了映射注解，用于将应用的领域类型映射到支撑的数据库结构之上。

在我们进一步探讨Cassandra之前，有一点很重要，那就是尽管Cassandra与关系数据库（如Oracle和SQL Server）有许多相似的概念，但Cassandra并不是关系数据库，在很多方面与关系数据库截然不同。我

将尝试解释Cassandra的独特之处，因为这与如何使用Spring Data有关。我鼓励你阅读Cassandra自己的文档，以全面了解Cassandra的工作原理。

下面我们从在Taco Cloud项目中启用Spring Data Cassandra开始。

12.2.1 启用Spring Data Cassandra

要开始使用Spring Data Cassandra的反应式repository功能，我们需要添加反应式Spring Data Cassandra的Spring Boot starter依赖。实际上，我们可以从两个Spring Data Cassandra starter依赖间进行选择。

如果不打算为Cassandra编写反应式repository，那么我们可以在构建文件中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-cassandra</artifactId>
</dependency>
```

这个依赖也可以在Initializr中通过选中Cassandra复选框添加进来。

在本章中，我们主要关注编写反应式repository，所以需要使用另外一个支持反应式Cassandra repository的starter依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    spring-boot-starter-data-cassandra-reactive
  </artifactId>
</dependency>
```

如果使用Spring Initializr创建项目，我们可以通过选中Reactive Cassandra复选框将这个依赖添加到构建文件中。

很重要的一点在于，我们使用这个依赖替代了Spring Data JPA starter依赖。此时我们不再通过JPA将数据持久化到关系型数据库中，而是使用Spring Data将数据持久化到Cassandra数据库中。因此，我们可能想要从构建文件中移除Spring Data JPA starter依赖和关系型数据库的依赖（如JDBC驱动和H2依赖）。

Spring Data Reactive Cassandra starter依赖会为项目引入多个依赖项，其中包括Spring Data Cassandra库和Reactor。由于这些库位于运行时类路径中，因此将会触发创建反应式Cassandra库的自动配置。这意味着我们马上就能开始编写反应式Cassandra repository，而无须太多显式配置。

不过，少量的配置还是需要的，至少需要配置键空间（key space）的名称，我们的repository要在该键空间中进行操作。为了做到这一点，我们先创建一个键空间。

> 注意：在Cassandra中，键空间是Cassandra节点中的一组表。这与关系数据库中表、视图和约束的分组方式大致类似。

尽管我们可以配置Spring Data Cassandra自动创建键空间，但是手动创建（或使用现有的键空间）通常要容易得多。借助Cassandra CQL

（Cassandra Query Language，Cassandra查询语言） shell，我们可以使用如下的create keyspace命令为Taco Cloud应用创建键空间：

```
cqlsh> create keyspace tacocloud
   ... with replication={'class':'SimpleStrategy', 'replication_factor':1}
   ... and durable_writes=true;
```

简而言之，这里创建了一个名为tacocloud的键空间，并且使用简单策略的复制（replication）和持久性写入（durable writes）。通过将复制因子设置为1，我们希望为每行数据保留一个副本。复制策略决定了该如何处理复制操作。SimpleStrategy复制策略对于单数据中心（和样例）使用来说是不错的选择，但是如果你的Cassandra集群跨多个数据中心，那就应该考虑使用NetworkTopologyStrategy。推荐你阅读一下Cassandra的文档，了解复制策略的更多细节以及创建键空间的其他可选项。

现在，我们已经创建了键空间，接下来应该配置spring.data.cassandra.keyspace-name属性，告诉Spring Data Cassandra该如何使用该键空间：

```
spring:
  data:
    cassandra:
      keyspace-name: tacocloud
      schema-action: recreate-drop-unused
```

在这里，我们将spring.data.cassandra.schema-action属性设置为recreate-drop-unused。这项配置在开发阶段非常有用，因为它会保证应用在每次重新启动的时候，所有的表和用户定义类型都将会删除并重建。它的默认值为none，不会对已有模式采取任何操作，在生产环境

中，这种设置是非常有用的，因为我们并不想在应用启动的时候删除所有生产环境中的表。

在本地运行Cassandra数据库时，我们只需要设置这两个属性。不过，除了这两个属性之外，你可能还想要设置其他的属性，这取决于你如何配置Cassandra集群。

默认情况下，Spring Data Cassandra会假定Cassandra在本地运行并监听9092端口。如果事实并非如此，那么在生产环境的配置中我们可能还要配置spring.data.cassandra.contact- points和spring.data.cassandra.port属性：

```
spring:
  data:
    cassandra:
      keyspace-name: tacocloud
      contact-points:
      - casshost-1.tacocloud.com
      - casshost-2.tacocloud.com
      - casshost-3.tacocloud.com
      port: 9043
```

注意，spring.data.cassandra.contact-points属性是我们识别Cassandra主机名的地方。每个联系点（contact point）代表了运行Cassandra节点的主机。默认情况下，它会被设置为localhost，但是我们可以将其设置为主机名的一个列表。应用会尝试连接每个连接点，直到能够连接上其中的一个为止。这样能够确保在Cassandra集群中不会出现单点故障，应用能够通过给定的连接点与集群建立连接。

我们可能还需要设置Cassandra集群的用户名和密码。这可以通过设

置spring.data.cassandra.username和spring.data.cassandra.password属性来实现：

```
spring:
  data:
    cassandra:
      ...
      username: tacocloud
      password: s3cr3tP455w0rd
```

现在，在我们的项目中已经启用和配置好了Spring Data Cassandra，接下来就应该将领域模型与Cassandra表进行映射并编写repository了。在此之前，我们回过头来看一些Cassandra数据模型的基本要点。

12.2.2 理解Cassandra的数据模型

正如前文所述，Cassandra与关系型数据库有很大的不同。在将领域类型映射为Cassandra表之前，理解Cassandra数据模型与关系型数据库数据持久化建模的差异是非常重要的。

关于Cassandra数据模型，有几项很重要的事情需要理解。

- Cassandra表可能有任意数量的列，但是并不是所有的行都会用到这些列。
- Cassandra数据库被分割为多个分区。给定表中的任何一行都可以由一个或多个分区管理，但是不太可能每个分区都拥有所有的行。
- Cassandra表有两种键：分区键（partition key）和集群键（clustering key）。Cassandra会对每一行的分区键执行哈希操作，以确定由哪个分区管理该行。集群键决定了行在分区中维护的顺序（不一定是

它们在查询结果中出现的顺序）。

- Cassandra对读操作进行了极大的优化。因此，较为常见和推荐的做法是让表实现高度非规范化，并让数据跨多个表进行复制（比如，客户信息可能会保存在customer表中，同时也会复制到客户所创建的订单表中）。

需要说明一点，将Taco Cloud领域类型调整为使用Cassandra，并不是简单地将几个JPA注解替换为Cassandra注解就可以了。我们必须重新考虑如何对数据进行建模。

12.2.3　将领域对象映射为Cassandra持久化

在第3章中，我们为领域类型（Taco、Ingredient、Order等）添加了JPA规范提供的注解。这些注解会将领域类型映射为要持久化到关系型数据库中的实体。尽管这些注解无法用于Cassandra的持久化，但是Spring Data Cassandra提供了自己的映射注解以达到同样的目的。

我们首先从Ingredient开始，它可以非常容易地映射到Cassandra上。如下是支持Cassandra的新Ingredient类：

```
package tacos;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Table("ingredients")
```

```
public class Ingredient {

  @PrimaryKey
  private final String id;
  private final String name;
  private final Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

看上去，Ingredient类与我前面所说的只需替换几个注解就可以的说法相矛盾。在这里，我们不再使用JPA持久化中的@Entity注解，而是使用了@Table注解，这表明配料将会持久化到名为ingredients的表中。另外，我们不再为id属性使用@Id，而是使用@PrimaryKey。到现在为止，我们似乎只是替换了几个注解而已。

但是，不要让Ingredient的映射欺骗了你。Ingredient是最简单的领域类型之一。如果我们将Taco类进行Cassandra持久化映射（如程序清单12.1所示），那就更有意思了。

程序清单12.1　为Taco类添加注解实现Cassandra持久化

```
package tacos;
import java.util.Date;
import java.util.List;
import java.util.UUID;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import org.springframework.data.cassandra.core.cql.Ordering;
import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;
import org.springframework.data.cassandra.core.mapping.Table;
import org.springframework.data.rest.core.annotation.RestResource;
import com.datastax.driver.core.utils.UUIDs;
```

```
import lombok.Data;

@Data
@RestResource(rel="tacos", path="tacos")
@Table("tacos")                                    ←--- 持久化到tacos表
public class Taco {

  @PrimaryKeyColumn(type=PrimaryKeyType.PARTITIONED)       ←--- 定义分区
键
  private UUID id = UUIDs.timeBased();

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  @PrimaryKeyColumn(type=PrimaryKeyType.CLUSTERED,       ←--- 定义集群键
                    ordering=Ordering.DESCENDING)
  private Date createdAt = new Date();

  @Size(min=1, message="You must choose at least 1 ingredient")
  @Column("ingredients")                                  ←--- 将列表映射到in
gredients列
  private List<IngredientUDT> ingredients;

}
```

我们可以看到，Taco类的映射会更加复杂。与Ingredient类似，它也使用@Table注解声明taco应该写入到名为tacos的表中。但是，这是它与Ingredient唯一的相似之处。

id属性依然是主键，但它只是两个主键列中的一个而已。具体来讲，id属性使用了@PrimaryKeyColumn注解，并且type的值为PrimaryKeyType.PARTITIONED。这表明id属性要作为分区键，用来确定taco数据的每一行要写入到哪个分区中。

你可能也会发现，id属性现在是UUID类型，而不是Long类型。虽然不是强制要求，但是保存系统生成的ID值的属性通常是UUID类型

的。此外，针对新Taco对象，这里的UUID会使用基于时间的UUID进行初始化（但是，从数据库中读取已有Taco时，它可能会被覆盖）。

我们继续往下看，createdAt属性映射到了另外一个主键列。但是，在本例中，@PrimaryKeyColumn的type属性设置成了PrimaryKeyType.CLUSTERED，这意味着createdAt会作为集群键。按照前文所述，集群键用来确定行在集群中的顺序。更具体来讲，我们将顺序设置为降序，所以，在给定的分区中，较新的行会优先出现在taco表中。

最后，ingredients属性是一个IngredientUDT对象的List，而不再是Ingredient对象的List。Cassandra表是高度非规范化的，因此可能会包含与其他表重复的数据。尽管ingredient表代表了所有可用配料的记录，但是taco所选择的配料会重复保存到ingredients列中。我们不会简单地引用ingredients表中的一行或多行，而是会让ingredients属性包含所有已选配料的完整数据。

但是，我们为什么会引入新的IngredientUDT类呢？为何不重用Ingredient类呢？简而言之，包含数据集合的列，比如ingredients列，必须是原生类型（整型、字符串等）的集合或用户定义类型（user-defined type）的集合。

在Cassandra中，用户定义类型能够让我们声明比原生类型更丰富的表的列。通常，它们会作为关系型结构中外键的非规范化模拟形式。但是，外键只是引用另外一张表中的一行数据，与之不同，用户定义类型

实际上会持有其他表中某行数据的副本。在tacos表的ingredients列中，它将会包含配料定义的数据结构集合。

我们不能将Ingredient用作用户定义类型，因为@Table注解已经将其映射成了Cassandra中的一个持久化实体。所以，我们必须创建一个新的类，定义该如何将配料信息存储到taco表的ingredients列上。IngredientUDT类（其中UDT代表了用户定义类型，即user-defined type）就是完成这项工作的：

```
package tacos;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@UserDefinedType("ingredient")
public class IngredientUDT {

  private final String name;
  private final Ingredient.Type type;

}
```

尽管IngredientUDT和Ingredient看上去非常相似，但是它的映射需求要简单得多。它使用了@UserDefinedType注解，表明这是Cassandra中的用户定义类型。但是就其他方面来讲，它就是有几个属性的简单类。

我们会发现，IngredientUDT类没有包含id属性。尽管它也可以包含

源Ingredient中id属性的副本，但是这样没有太大必要。实际上，用户定义类型可以包含任何想要的属性，它没有必要与表定义一一对应。

我发现，可视化用户定义类型与表中的持久化数据之间的关联关系是很困难的。图12.1展现了整个Taco Cloud数据库的数据模型，包含了用户定义类型。

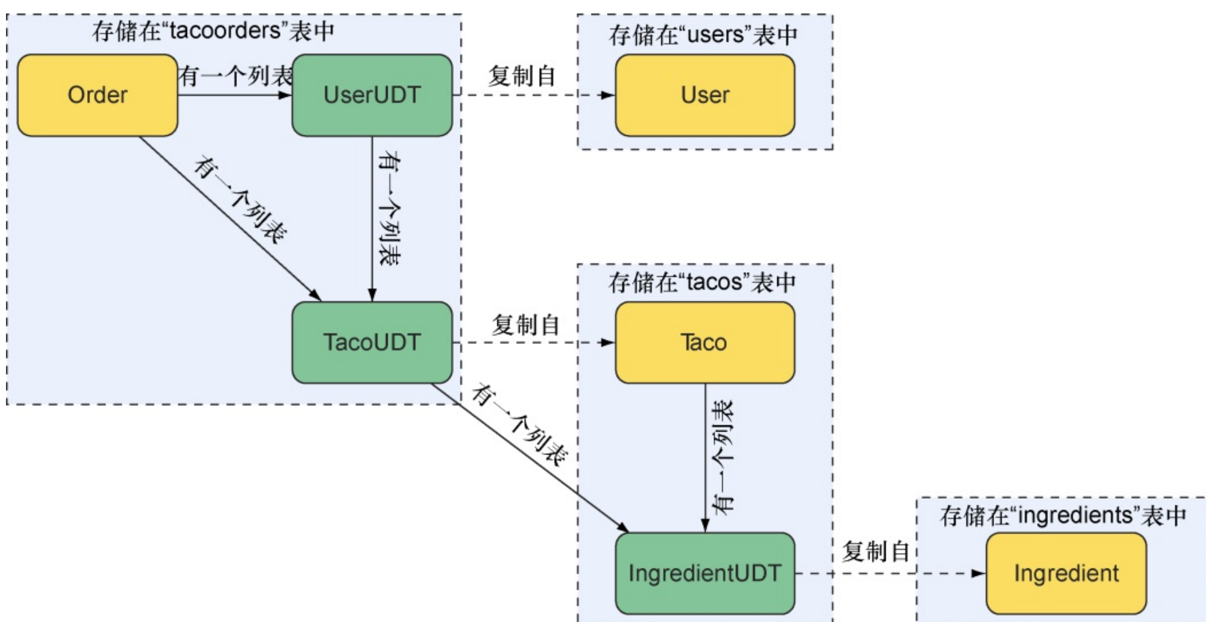


图12.1　在这里不再使用外键和连接，Cassandra表是非规范化的，用户定义类型包含从关联表复制的数据

具体到我们刚刚创建的用户定义类型，需要注意Taco有一个IngredientUDT列表，其中包含了从 Ingredient 复制而来的数据。当 Taco 持久化的时候，Taco 对象以及IngredientUDT 列表都会持久化到 tacos 表中。IngredientUDT列表会完整地持久化到ingredients列中。

另外一种帮助我们理解用户定义类型如何使用的办法就是从数据库中查询tacos表的各个行。借助Cassandra提供的CQL和cqlsh工具，我们

可以看到如下的结果：

```
cqlsh:tacocloud> select id, name, createdAt, ingredients from tacos;

id         | name      | createdat | ingredients
----------+-----------+-----------+----------------------------------------
 827390...| Carnivore | 2018-04...| [{name: 'Flour Tortilla', type: 'WRAP'
},
                                     {name: 'Carnitas', type: 'PROTEIN'},
                                     {name: 'Sour Cream', type: 'SAUCE'},
                                     {name: 'Salsa', type: 'SAUCE'},
                                     {name: 'Cheddar', type: 'CHEESE'}]
(1 rows)
```

从中可以看到，id、name和createdat列包含的都是简单的值。在这方面，它们与关系数据库的类似查询差别不大。ingredients列就不一样了，按照定义，它包含用户定义的ingredient类型（由IngredientUDT所定义）的集合，所以它的值显示为一个JSON数组，数组中则是JSON对象。

你可能也注意到了在图12.1中还有其他的用户定义类型。在继续将领域对象映射为Cassandra的过程中，我们肯定要创建更多的用户定义类型，其中包括Order类所用到的类型。程序清单12.2显示的Order类，针对Cassandra持久化进行了修改。

程序清单12.2　将Order类映射为Cassandra tacoorders表

```
@Data
@Table("tacoorders")                                 ←--- 映射到tacoorders表
public class Order implements Serializable {

  private static final long serialVersionUID = 1L;

  @PrimaryKey                                    ←--- 声明主键
  private UUID id = UUIDs.timeBased();
```

```
  private Date placedAt = new Date();

  @Column("user")                     ←--- 映射到user列
  private UserUDT user;

  // delivery and credit card properties omitted for brevity's sake

  @Column("tacos")                                          ←--- 将一个列表映射
到tacos列
  private List<TacoUDT> tacos = new ArrayList<>();

  public void addDesign(TacoUTD design) {
    this.tacos.add(design);
  }

}
```

程序清单12.2有意省略了Order的许多属性，这些属性不适合Cassandra数据建模的讨论。剩下的属性和映射方式类似于Taco的定义。就像以前使用@Table一样，在这里@Table用于将Order映射到tacoorders表。在本例中，我们不关注顺序，因此id属性只使用了@PrimaryKey注解，将其同时作为分区键和集群键，并采用了默认的排序。

tacos属性比较有趣，因为它是List<TacoUDT>，而不是Taco对象的列表。在这里，Order和Taco/TacoUDT之间的关系类似于前文中Taco和Ingredient/IngredientUDT之间的关系。也就是说，我们不是通过外键将不同表中的多行数据关联在一起，而是让Order表包含所有的taco数据，以便于优化表的快速读取。

类似的，user属性引用了UserUDT对象，它会持久化到user列中。同样，这与关系型数据库中连接另外一张表的策略是不同的。

至于TacoUDT，它与IngredientUDT类非常相似，不过它里面包含

了对另外一个用户定义类型的引用：

```
@Data
@UserDefinedType("taco")
public class TacoUDT {

  private final String name;
  private final List<IngredientUDT> ingredients;

}
```

UserUDT更有趣一点，因为它包含了3个属性，而不是两个：

```
@UserDefinedType("user")
@Data
public class UserUDT {

  private final String username;
  private final String fullname;
  private final String phoneNumber;

}
```

如果能够重用第3章定义的领域类或者仅仅将JPA注解替换为Cassandra注解，那当然很好，但是Cassandra持久化的本质特点是要求我们重新思考数据该如何建模。现在，我们已经映射好了领域模型，接下来该编写repository了。

12.2.4　编写反应式Cassandra repository

正如我们在第3章所看到的，使用Spring Data编写repository只需声明一个接口，让它扩展Spring Data的基础repository，并有选择性地声明用于自定义查询的方法即可。实际上，编写反应式repository并没有太大

的不同。主要区别在于，我们需要扩展一个不同的基础repository接口，而且我们的方法将会处理反应式发布者，如Mono和Flux，而不再是领域类型和集合。

在编写反应式Cassandra repository时，我们有两个基础接口可选：ReactiveCassandraRepository和ReactiveCrudRepository。选择哪个接口很大程度上取决于该如何使用repository。ReactiveCassandraRepository扩展了ReactiveCrudRepository，提供了insert()方法的一些变种，如果要保存的对象是新建的，这些变种进行了优化。除此之外，ReactiveCassandraRepository提供了与ReactiveCrudRepository相同的操作。如果我们想要插入很多数据，那么可能需要选择ReactiveCassandraRepository；否则，最好选择ReactiveCrudRepository，因为在不同数据库类型之间它更具可移植性。

Cassandra repository必须是反应式的吗？

尽管我们本章主要关注如何使用Spring Data编写反应式repository，但是你可能想知道该如何为Cassandra编写非反应式的repository。如果是这样，那么我们需要让repository接口扩展非反应式的CrudRepository或CassandraRepository接口，而不是扩展ReactiveCrud Repository或ReactiveCassandraRepository。我们的repository方法就可以返回带有Cassandra相关注解的领域类型或这些领域类型的集合，而不再是Flux和Mono。

如果你准备采用非反应式的repository，那么可以将starter依赖从spring-boot- starter-data-cassandra-reactive修改为spring-boot-starter-data-cassandra，不过这并不是严格要求的。

重新看一下我们为Taco Cloud编写的repository，要使它们变成反应式的，我们首先让它们扩展ReactiveCrudRepository 或ReactiveCassandraRepository，而不再是CrudRepository。我们首先看一下IngredientRepository。除了使用配料数据初始化数据库之外，我们不会插入很多的新配料数据。所以，IngredientRepository可以扩展ReactiveCrudRepository，如下所示：

```
public interface IngredientRepository
         extends ReactiveCrudRepository<Ingredient, String> {
}
```

我们不需要在IngredientRepository中定义任何的自定义查询，所以要将IngredientRepository变成反应式repository，并不需要额外的工作。现在，它扩展了ReactiveCrud Repository，所以它的方法处理的都是Flux和Mono。例如，findAll()方法现在返回的是Flux<Ingredient>，而不是Iterable<Ingredient>。所以，在使用它的时候，要按照正确的方式来使用。比如，IngredientController需要重写为返回Flux<Ingredient>：

```
@GetMapping
public Flux<Ingredient> allIngredients() {
  return repo.findAll();
}
```

TacoRepository的变更要稍微复杂一些。我们不用像非反应式repository那样扩展PagingAndSortingRepository，而是可以扩展ReactiveCassandraRepository。在参数化Taco对象的时候，不能使用Long类型的ID属性，在与Taco对象协作的时候，要使用UUID类型的ID：

```
public interface TacoRepository
        extends ReactiveCrudRepository<Taco, UUID> {
}
```

因为这个新TacoRepository的findAll()方法会返回Flux<Ingredient>，所以我们不用让它扩展PagingAndSortingRepository，也不用操作分页的数据。相反，在DesignTacoController的recentTacos()方法中，我们只需要调用返回的Flux的take()方法来限制要消费的Taco对象的数量即可（实际上，在11.1.2节中，我们已经修改了DesignTacoController和它的recentTacos()方法）。

OrderRepository所需的变更也很简单。我们不再扩展CrudRepository，而是让它扩展ReactiveCassandraRepository：

```
public interface OrderRepository
        extends ReactiveCassandraRepository<Order, UUID> {
}
```

最后，我们来看一下UserRepository。我们可能还记得，UserRepository有一个自定义的查询方法，即findByUsername()。这个方法让定义Cassandra持久化repository有了一些变化。支持Cassandra的UserRepository代码如下：

```
public interface UserRepository
```

```
        extends ReactiveCassandraRepository<User, UUID> {

  @AllowFiltering
  Mono<User> findByUsername(String username);

}
```

与其他的repository接口（除了IngredientRepository）类似，UserRepository也扩展了ReactiveCassandraRepository。到目前为止，没有感到惊讶的地方。但是，它的findByUsername()方法我们需要注意一下。

首先，因为这是一个反应式repository，所以findByUsername()不会再简单地返回User对象。我们对其进行了重新定义，让它返回Mono<User>。一般而言，在反应式repository中，我们自定义的查询方法应该要么返回Mono（要返回的值不超过一个），要么返回Flux（会有多个返回值）。

同时，按照Cassandra的特点，在查询表的时候，我们不能像在关系型数据库的SQL中那样简单地使用where子句。Cassandra对读取进行了优化，但是使用where子句进行过滤可能会拖慢其他快速查询的速度。即便如此，根据一个或多个列对表进行查询还是非常有用的。因此，@AllowFiltering注解使结果的过滤变成了现实，它可以作为这些场景的可用方案。

在findByUsername()中，我们预期的CQL查询如下所示：

```
select * from users where username='some username';
```

同样，Cassandra是不允许这样做的。但是，在将@AllowFiltering注解放到findByUsername()方法上之后，所形成的CQL查询如下所示：

```
select * from users where username='some username' allow filtering;
```

查询末尾的allow filtering子句提醒Cassandra，我们已经意识到查询性能的潜在影响，并且无论如何都需要它。在这种情况下，Cassandra将允许使用where子句并按需过滤结果。

Cassandra中有很多强大功能，当它与Spring Data和Reactor结合使用时，我们可以在Spring应用中充分使用这些功能。但是，让我们把注意力转移到支持反应式repository的另一个数据库上来，那就是MongoDB。

12.3 编写反应式的MongoDB repository

MongoDB是另一个知名的NoSQL数据库。Cassandra是行存储数据库，而MongoDB则被视为文档数据库。更具体来讲，MongoDB以BSON（Binary JSON，二进制JSON）格式存储文档，我们可以使用与查询其他数据库中的数据类似的方式查询和检索文档。

与Cassandra一样，必须要明确知道MongoDB不是关系数据库。管理MongoDB服务器集群和数据建模的方式与处理其他类型数据库时的思维方式是不一样的。

不过，使用MongoDB和Spring Data与使用Spring Data处理JPA或

Cassandra并没有太大的差异。我们会在领域类上使用注解，将领域类型映射为文档结构。我们还会编写repository接口，这遵循与JPA和Cassandra一样的编程模型。但是在进行任何操作之前，我们必须在项目中启用Spring Data MongoDB。

12.3.1　启用Spring Data MongoDB

要启用Spring Data MongoDB，我们需要将Spring Data MongoDB starter添加到项目的构建文件中。Spring Data MongoDB有两个独立的可选starter。

如果你使用非反应式的MongoDB，那么需要将如下的依赖添加到构建文件中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    spring-boot-starter-data-mongodb
  </artifactId>
</dependency>
```

这项依赖也可以在Spring Initializr中通过选中名为MongoDB的复选框添加进来。但是，本章主要关注的是编写反应式repository，所以我们要选择反应式Spring Data MongoDB starter依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    spring-boot-starter-data-mongodb-reactive
  </artifactId>
</dependency>
```

在Initializr中，我们可以通过选中Reactive MongoDB复选框将反应式Spring Data MongoDB starter添加进来。将这个starter添加到构建文件中之后，自动配置功能将会触发，启用Spring Data对自动化repository接口的支持，这一点与第3章的JPA和第11章的Cassandra类似。

默认情况下，Spring Data MongoDB会假定MongoDB在本地运行并监听27017端口。为了测试和开发的便利性，我们可以选择使用嵌入式的Mongo数据库。为了实现这一点，我们需要将Flapdoodle Embedded MongoDB依赖添加到构建文件中：

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
</dependency>
```

与我们在关系型数据库中使用H2类似，Flapdoodle嵌入式数据库带来了使用内存Mongo数据库的便利性。也就是说，我们不需要运行单独的数据库，但是所有的数据会在应用重启的时候丢掉。

嵌入式数据库对于开发和测试是很不错的，一旦我们将应用部署到生产环境，就需要设置几个属性，让Spring Data MongoDB知道访问何处的Mongo数据库以及该如何进行访问：

```
spring:
  data:
    mongodb:
      host: mongodb.tacocloud.com
      port: 27018
      username: tacocloud
      password: s3cr3tp455w0rd
      database: tacoclouddb
```

在这里，并不是所有的属性都是必需的。如果Mongo数据库不在本地运行，那么这些属性能够为Spring Data MongoDB指明正确的方向。拆分一下上面的配置，如下就是要设置的每个属性。

- spring.data.mongodb.host：Mongo运行的主机名（默认为localhost）。
- spring.data.mongodb.port：Mongo服务器监听的端口（默认为27017）。
- spring.data.mongodb.username：访问安全Mongo数据库的用户名。
- spring.data.mongodb.password：访问安全Mongo数据库的密码。
- spring.data.mongodb.database：数据库名（默认为test）。

在我们的项目中，已经启用了Spring Data MongoDB，所以接下来我们需要为领域对象添加注解，以便于将它们持久化为MongoDB中的文档。

12.3.2 将领域对象映射为文档

Spring Data MongoDB提供了多个注解。在将领域对象映射为要持久化到MongoDB中的文档结构时，这些注解是非常有用的。尽管Spring Data MongoDB提供了多个用于映射的注解，但是其中的3个是最常用的。

- @Id：将某个属性指明为文档的ID（来自Spring Data Commons）。
- @Document：将领域类型声明为要持久化到MongoDB中的文档。
- @Field：指定某个属性持久化到文档中的字段名称（以及可选的顺序配置）。

在这3个注解中，@Id和@Document是严格需要的。除非显式指定，否则没有使用@Field注解的属性将假定字段名与属性名相同。

将这些注解应用到Ingredient类上的效果如下所示：

```
package tacos;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document
public class Ingredient {

  @Id
  private final String id;
  private final String name;
  private final Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

可以看到，我们在类级别使用了@Document注解，表明Ingredient是一个文档实体，可以在Mongo数据库中执行读取和写入操作。默认情况下，集合名（这是Mongo中与关系型数据库的表对等的概念）是基于类名的，只不过第一个字母会变成小写。因为我们没有特别指定，所以Ingredient对象将会持久化到名为ingredient的集合中。但是，我们可以通过设置@Document的collection属性改变这种行为：

```
@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document(collection="ingredients")
public class Ingredient {
...
}
```

我们还会看到，id属性使用了@Id注解。这表明该属性将会作为要持久化的文档的ID。我们可以将@Id注解用到任意Serializable类型的字段上，包括String和Long。在本例中，我们已经使用String定义的id属性作为自然标识符，因此不需要将其更改为其他类型。

到目前为止，一切都很顺利。但是，不要忘了，在本章前面的内容中，我们曾说过Ingredient是进行Cassandra映射时最简单的一个领域类型。其他的类型，比如Taco，就稍微困难一些了。接下来，我们看一下如何映射Taco类，看看它会有哪些惊喜。

在将领域类型映射为MongoDB文档时，我们肯定需要为Taco添加@Document注解。同时，我们还需要通过@Id注解指定ID属性。在添加完支持MongoDB持久化的注解后，我们就会得到如下的Taco类：

```
@Data
@RestResource(rel="tacos", path="tacos")
@Document
public class Taco {

  @Id
  private String id;

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  private Date createdAt = new Date();
```

```
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<Ingredient> ingredients;

}
```

不管你是否相信，这就是所有的内容。在Cassandra中，我们还需要处理两个不同的主键字段并且要引用用户定义类型，但这是Cassandra特有的。对于MongoDB来说，Taco的映射要简单得多。

即便如此，在Taco中还是有一些有意思的事情值得关注。首先，我们要注意，id属性变成了String类型（而不是JPA版本中的Long类型或Cassandra版本中的UUID类型）。正如我在前文所述，@Id注解可以用到任意Serializable类型上。如果选择使用String属性作为ID，我们就可以在保存的时候让Mongo自动设置一个值给它。将其设置为String类型之后，我们就得到了一个数据库管理赋值的ID，而不用再担心如何手动设置该属性。

我们再看一下ingredients属性。它是一个List<Ingredient>，与第3章中的JPA版本非常类似。与JPA版本不同的是，这个列表不会存储到单独的MongoDB集合中。与Cassandra对应的功能类似，配料列表会直接、以非规范化的形式存储到taco文档中。不过，与Cassandra不同，我们不需要创建用户定义类型，MongoDB非常乐意使用任何类型，不管它是带有@Document注解的另一个类型还是简单的POJO，都是可以的。

看到将Taco映射为文档持久化非常容易，我们可以松口气了。这种映射的便利性会延续到Order领域类吗？你可以自行看一下带有

MongoDB注解的Order类：

```
@Data
@Document
public class Order implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  private String id;

  private Date placedAt = new Date();

  @Field("customer")
  private User user;

  // other properties omitted for brevity's sake

  private List<Taco> tacos = new ArrayList<>();

  public void addDesign(Taco design) {
    this.tacos.add(design);
  }

}
```

简单起见，我删除了投递和信用卡相关的各种字段。从剩下的部分可以清楚地看出，与其他领域类型一样，我们只需要@Document和@Id注解。即便如此，我们也为user属性使用了@Field，指定在持久化文档中它将会存储为customer。

User领域类的MongoDB持久化映射依然非常简单，看到这里，相信你并不会对此感到意外：

```
@Data
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@RequiredArgsConstructor
@Document
public class User implements UserDetails {
```

```
  private static final long serialVersionUID = 1L;

  @Id
  private String id;

  private final String username;

  private final String password;
  private final String fullname;
  private final String street;
  private final String city;
  private final String state;
  private final String zip;
  private final String phoneNumber;

  // UserDetails method omitted for brevity's sake

}
```

虽然有一些更高级和不常见的场景需要额外的映射，但是我们会发现，对于大多数情况，@Document、@Id以及偶尔用到的@Field对于MongoDB映射来说已经足够了。对于Taco Cloud的领域类型，它们完全可以胜任。

剩下的事情就是编写repository接口了。

12.3.3　编写反应式的MongoDB repository接口

Spring Data MongoDB提供的自动化repository功能与Spring Data JPA和Spring Data Cassandra类似。在为MongoDB编写反应式repository的时候，我们可以在ReactiveCrudRepository和ReactiveMongoRepository之间进行选择。核心的差异在于，ReactiveMongoRepository提供多个特殊的insert()方法，它们针对新文档的持久化进行了优化，而

ReactiveCrudRepository依赖save()方法来保存新文档和已有的文档。

如何编写非反应式的**MongoDB repository**？

本章主要关注如何使用Spring Data编写反应式的repository。如果出于某种原因，你希望使用非反应式的repository，那么可以通过让repository接口扩展CrudRepository或MongoRepository来实现，而不是选择扩展ReactiveCrudRepository或ReactiveMongo Repository。这样，我们就可以让repository返回带有Mongo注解的领域类型或这些领域类型的集合。

尽管不是严格要求的，但是你可以将spring-boot-starter-data-mongodb-reactive依赖替换为spring-boot-starter- data-mongodb。

首先，我们来定义将Ingredient对象持久化为文档的repository。在数据库初始化完成之后，我们不会频繁地创建配料的文档，甚至有可能永远不会这样做。因此，ReactiveMongoRepository提供的优化没有太多的用处，我们可以让IngredientRepository扩展ReactiveCrudRepository：

```
package tacos.data;
import org.springframework.data.repository.reactive.ReactiveCrudRepository
;
import org.springframework.web.bind.annotation.CrossOrigin;
import tacos.Ingredient;

@CrossOrigin(origins="*")
public interface IngredientRepository
         extends ReactiveCrudRepository<Ingredient, String> {
}
```

稍等片刻！它看起来与我们在12.2.4小节中为Cassandra编写的IngredientRepository接口是完全一样的！实际上，这是同一个接口，没有任何变化。这凸显了扩展ReactiveCrudRepository的一个好处，也就是它在各种数据库类型之间具有更强的可移植性，并且针对MongoDB和Cassandra都可以很好地运行。

因为它是一个反应式repository，所以它的方法处理的是Flux和Mono，而不是原始领域类型或这些领域类型的集合。例如，findAll()方法将返回Flux<Ingredient>，而不是Iterable<Ingredient>。同样，findById()将返回Mono<Ingredient>，而不是Optional <Ingredient>。因此，这个反应式repository可以作为端到端反应式流的一部分。

现在，为了将Taco持久化为MongoDB中的文档，我们定义另一个repository。与配料文档不同，我们会频繁创建taco文档。因此，ReactiveMongoRepository优化过的insert()方法就很有价值了。如下的代码片段展现了支持MongoDB的TacoRepository接口：

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository
;
import reactor.core.publisher.Flux;
import tacos.Taco;
public interface TacoRepository
         extends ReactiveMongoRepository<Taco, String> {

  Flux<Taco> findByOrderByCreatedAtDesc();

}
```

相对于ReactiveCrudRepository，使用ReactiveMongoRepository唯一

的缺点在于它是专属于MongoDB的，不能迁移至其他数据库。在你的项目中，你需要确定这种代价是否值得。如果你预计不会在某个时刻切换到不同的数据库，那么尽可以选择ReactiveMongoRepository并充分利用它针对数据插入操作所带来的优化。

注意，在TacoRepository中，我们引入了一个新的方法。这个方法支持显示最近创建的taco。在JPA版本的repository中，我们需要通过扩展PagingAndSortingRepository实现该功能。但是，在反应式repository中，PagingAndSortingRepository并没有太大的用处（尤其是分页功能）。在Cassandra版本中，排序是通过表定义中的集群键实现的，所以在repository中获取最近创建的taco时，我们并不需要特殊的处理。

对于MongoDB来说，我们想要获取最近创建的taco。尽管名字看上去有些奇怪，但是findByOrderByCreatedAtDesc()方法遵循自定义查询方法命名约定。它说明我们想要查找Taco对象，没有任何查询条件，我们在这里没有设置任何必须匹配的属性。然后，我们告诉它将结果按照createdAt属性降序排列。

在这里，命名中使用空By子句的原因在于方法名称中还有另一个By，这样做可以避免方法名称出现误解。如果将其命名为findAllOrderByCreatedAtDesc()，那么名称中的AllOrder部分将被忽略，Spring Data将尝试通过匹配createdAtDesc属性来查找taco。因为不存在该属性，所以应用将会报错，无法正常启动。

因为findByOrderByCreatedAtDesc()返回的是一个Flux<Taco>，所以

我们不用担心分页的事情。相反，我们只需要使用take操作获取Flux发布的前12个Taco即可。例如，在显示最近创建的taco的控制器中，我们可以按照如下方式调用findByOrderBy CreatedAtDesc()：

```
Flux<Taco> recents = repo.findByOrderByCreatedAtDesc()
                         .take(12);
```

最终得到的Flux所发布的Taco条目不会超过12个。

再看OrderRepository接口，它非常简单：

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository
;
import reactor.core.publisher.Flux;
import tacos.Order;
public interface OrderRepository
         extends ReactiveMongoRepository<Order, String> {

}
```

我们会频繁创建Order文档，所以OrderRepository扩展了ReactiveMongoRepository，从而充分利用其insert()方法所带来的优化。除此之外，相对于我们已经定义的repository，它并没有什么新奇之处。

最后，我们看一下将User对象持久化为文档的repository：

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository
;
import reactor.core.publisher.Mono;
import tacos.User;

public interface UserRepository
         extends ReactiveMongoRepository<User, String> {
```

```
  Mono<User> findByUsername(String username);
}
```

讲解到现在，你对这个repository接口应该没有丝毫感到惊讶的地方了。与其他repository类似，它扩展了ReactiveMongoRepository（当然，它也可以扩展ReactiveCrudRepository）。唯一的与众不同之处在于，它有一个findByUsername()方式，这是在第4章中我们为了支持认证功能添加上去的。在这里，将它修改为返回Mono<User>，而不是原始的User对象。

12.4　小结

- Spring Data支持为Cassandra、MongoDB、Couchbase和Redis数据库创建反应式repository。
- Spring Data的反应式repository遵循与非反应式repository相同的编程模型，只不过它们所处理的是反应式发布者，如Flux和Mono。
- 非反应式repository（比如JPA repository）可以调整为使用Mono和Flux，但是在保存和获取数据时它们依然是阻塞的。
- 在使用非关系数据库时，需要理解如何恰当地为数据建模，这个建模过程决定了数据库最终如何存储数据。

---

[1]　Spring Data R2DBC致力于解决关系型数据库的反应式访问问题。——译者注

第4部分　云原生Spring

第4部分将会拆分单体应用模型，我们会介绍Spring Cloud和微服务的开发。在第13章中，简单介绍微服务之后，我们将会深入介绍服务发现，这里会使用Spring和Netflix的Eureka服务注册中心实现基于Spring的微服务的注册和发现。第14章通过Spring Cloud的Config Server探讨中心化的配置，Config Server服务能够为应用中的所有微服务提供中心化的配置。在第15章中，我们将会借助Netflix Hystrix实现断路器模式，让服务面对失败时更具弹性。

第13章　注册和发现服务

本章内容：

- 思考微服务
- 创建服务注册中心
- 注册和发现服务

你看过《海底总动员》（*Finding Nemo*）吗？在这部电影中，马林（小丑鱼）和多莉（蓝唐王鱼）试图去澳大利亚悉尼寻找马林失踪的儿子尼莫。在路上，它们遇到了一群翻车鱼。为了好玩儿，这些翻车鱼把自己摆成了很多种形状——剑鱼、八眼鱼，它们甚至还摆成马林的样子来模仿它。当多莉问它们是否知道如何到达悉尼时，它们组成了悉尼歌剧院的形状，然后变成了一个指向东澳大利亚洋流的箭头。

虽然这部电影没有深入介绍每条翻车鱼的生活，但是我们可以假定每条鱼都是独立于其他翻车鱼的个体。它们都有自己的鳞片、鳍、鳃、眼睛、内脏，据我们所知，它们还有各自的希望和梦想。尽管如此，它

们还是一起努力形成这些有趣的形状，帮助马林和多莉前往澳大利亚。

本章我们将会讨论如何开发翻车鱼所组成的应用程序，这是一系列章节中的第一章。也就是说，你将会看到如何使用微服务（一些小的、独立的应用程序，它们协同工作以提供完整应用的功能）进行开发。

更具体地讲，我们将会看到如何使用Spring Cloud套件中一些最有用的组件，包括配置管理、容错以及本章的主题即服务发现。但是，在此之前，我们快速、整体地了解一下使用微服务开发意味着什么以及它们能够提供哪些收益。

13.1 思考微服务

到目前为止，我们都是将Taco Cloud开发为单个应用程序，它会构建为一个可部署的JAR或WAR。单个可部署的文件似乎是一种很自然的选择。毕竟，几十年来，大多数的应用程序都是这样部署的。即便可能会将应用程序拆分为多个模块进行构建，但最终我们还是形成一个JAR或WAR，并将其投入到生产环境之中。

在构建小型、简单应用程序的时候，这当然是显而易见的方式。有意思的是，小型应用程序往往会不断增长。当需要新特性的时候，我们能够轻而易举地向项目中添加更多的代码。在我们发觉之前，它已经变成了一个复杂的单体应用，甚至有自己的思想。就像电影《小鬼怪》（*Gremlins*）里的Mogwai一样，如果你一直喂它，它最终会变成一个与你作对的怪物[1]。

单体应用看似简单，但是它会面临各种挑战，如下所示。

- 单体应用难以理解：代码库越大，理解每个组件在整个应用程序中所担任的角色就越困难。
- 单体应用难以测试：随着应用的不断增长，全面的集成和验收测试会变得更加复杂。
- 单体应用更容易出现库冲突：实现某个特性所需要的依赖可能会与其他特定的依赖不兼容。
- 单体应用的扩展较为低效：如果处于扩展的目的要将应用程序部署到更多的硬件上，那么我们必须要将整个应用部署到更多的服务器上，即便应用程序中很小的一部分需要扩展也同样如此。
- 单体应用中的技术决策是针对整个单体应用的：当为应用程序选择语言、运行时平台、框架和库的时候，整个应用程序都会遵循我们的选择，即便我们所做的选择只是为了支持某个单独的用户场景时同样如此。
- 单体应用需要大量的操作过程才能投入生产环境：当应用程序只有一个部署单元时，似乎更容易将其投入生产环境。事实上并非如此，单体应用程序的规模和复杂性通常需要更严格的开发过程和更周全的测试周期，这样才能保证所部署的应用程序是高质量的，才能避免引入bug。

在过去的几年间，微服务架构的出现致力于解决这些挑战。简而言之，微服务架构是将应用程序分解为可独立开发和部署的小规模、微型应用的一种方式。这些微服务之间互相协作，以实现更大的应用程序的功能。与单体应用程序架构相比，微服务架构有以下特点。

- 微服务易于理解：每个微服务与应用程序的其他微服务之间有一个很小且有限的契约。因此，微服务更加专注于目标，作为一个单

元，微服务更易于理解。

- 微服务易于测试：事情越小，就越便于测试。当你思考单元测试、集成测试和验收测试的时候，这一点非常明显。它也适用于微服务与单体应用之间的测试。
- 微服务较少受到库不兼容的影响：因为每个微服务都有自己的构建依赖项的集合，而这些依赖项不会与其他的微服务共享，所以不太可能会出现库冲突的现象。
- 微服务能够独立扩展：如果指定的微服务需要更多的处理能力，那么内存分配和/或实例数量可以按比例增加，而不会影响整体应用中其他微服务的内存和实例数量。
- 每个微服务可以选择不同的技术：每个微服务可以选择完全不同的语言、平台、框架和库。实际上，某个使用Java编写的微服务与另一个使用C#编写的微服务进行协作是完全合理的[2]。
- 微服务可以更加频繁地发布到生产环境中：尽管微服务架构的应用是由许多微服务组成的，但是部署每个微服务的时候，并不需要其他的微服务都已经部署就绪。而且，因为它们更小、更集中、更易于测试，所以将微服务投入到生产环境不需要那么多的繁文缛节。从产生想法到将其投入生产的耗时可以用分钟和小时计量，而不是用周和月。

显然，微服务能够让事情变得更简单。但是公平地讲，微服务架构并不是免费的午餐。微服务架构是一种分布式架构，有自己需要应对的挑战，包括网络延迟。在迁移至微服务架构时，我们需要记住这一点，因为很多的远程调用会累积并降低应用的速度。

你还要考虑是否应该将应用构建为微服务，因为并不是所有的应用程序都需要这种架构，或者说能从这种架构中受益。如果你的应用相对比较小或者比较简单，那么最初最好依然采用单体架构。随着它的不断

发展，再考虑将其拆分为微服务。

在开发云原生、微服务架构的应用时，要考虑很多因素。本章和接下来的几章主要关注Spring Cloud所提供的技术，以开发由微服务组成的应用程序。如果你对深入研究云原生应用程序的设计和思想过程感兴趣，那么建议阅读Cornelia Davis的*Cloud Native*（Manning，2019）。

微服务架构所面临的另外一个常见挑战就是每个服务该如何知道它要协作的其他服务在哪里。这恰好是本章的主题。事不宜迟，我们马上看一下如何使用Spring Cloud搭建一个服务注册中心。

13.2　搭建服务注册中心

Spring Cloud是一个非常大的伞形项目，由多个独立的子项目组成，每个子项目都以某种形式支撑着微服务的开发。其中有一个子项目叫作Spring Cloud Netflix，它按照Spring的编码风格重新提供了Netflix的多个组件。在这些组件中包括了Netflix的服务注册中心Eureka。

Eureka赤裸裸的历史真相

Eureka这个词最初的含义是当人们找到或发现某件事情时所发出的欢呼。这使得Eureka非常适合用作服务注册中心的名称，微服务要借助注册中心实现彼此发现的功能。

据传说，Eureka最早是由希腊物理学家阿基米德发明的，他坐在浴缸里的时候发现了浮力的原理，于是他跳出浴缸，赤裸裸地跑回家，嘴

里喊着“Eureka！”。

关于阿基米德是否真的光着身子跑回家并大喊“Eureka！”还有一些争论，但无论如何，这个故事非常有意思。话说回来，我们倒是可以衣冠整洁地使用Eureka服务注册中心。

在微服务应用中，Eureka会担当所有服务的注册中心。Eureka本身也可以视为一个微服务，只不过在整体应用中它的目的是让其他的服务能够互相发现。

鉴于它在微服务应用中的角色，在创建需要注册的服务之前，我们最好搭建一个Eureka服务注册中心。为了理解Eureka的运行原理，我们可以参见图13.1所述的流动过程。

当服务实例启动的时候，它会按照名称将自己注册到Eureka中。在图13.1中，服务的名称为“some-service”。“some-service”可能会有多个完全等价的实例，但是在Eureka注册时，它们的名称是相同的。

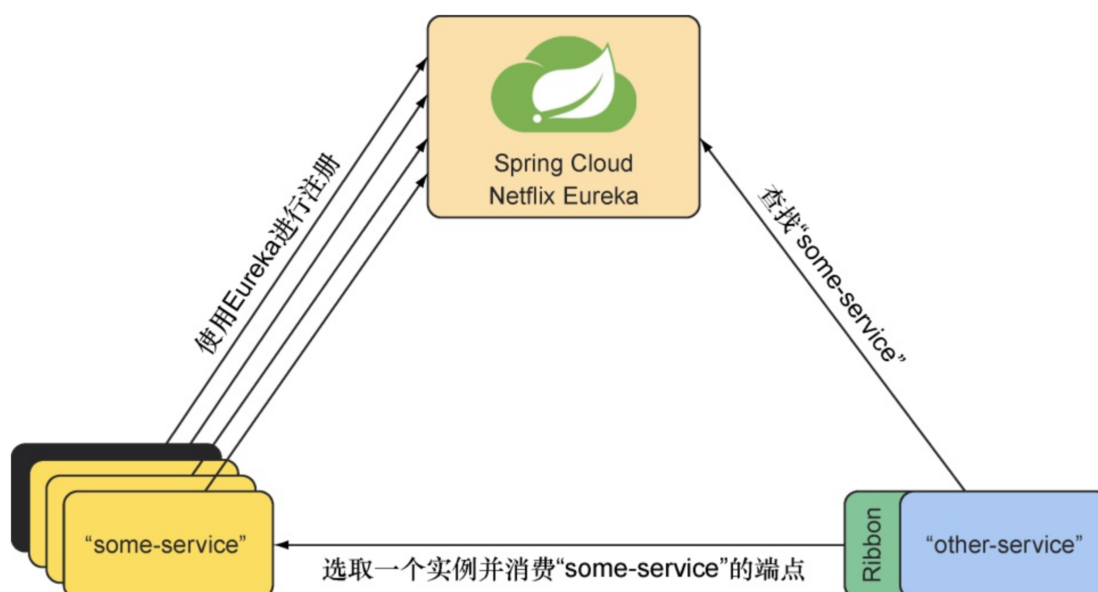


图13.1　服务使用Eureka服务注册中心进行注册（这样其他的服务就能发现并消费它们了）

在某个时间点，另一个服务（图13.1中名为“other-service”）需要使用“some-service”的端点。在这里，“other-service”没有使用特定的主机和端口信息对“some-service”进行硬编码，而是根据名字从Eureka查找“some-service”。Eureka的回应中将会包含它所知道的“some-service”的所有实例。

现在，“other-service”需要做决策了。它该使用“some-service”的哪个实例呢？如果它们都是完全等同的，其实就没什么关系了。为了避免每次都选择同一个实例，最好用一些客户端负载平衡算法来分散请求。这就是Netflix的另一个项目Ribbon的用武之地了。

虽然“other-service”完全可以自行查找和选择“some-service”的实例，但在这里我们让它依赖Ribbon。Ribbon是一个客户端负载平衡器，会帮助“other-service”做出选择。Ribbon做完选择之后，剩下的就是让“other-service”向Ribbon选择的实例发出请求。

为何要使用客户端负载均衡器

通常，我们会认为负载均衡器是一个中心化的服务，它处理所有的请求并将请求分发到多个目标实例中。与之不同，Ribbon是一个客户端负载均衡器，它会在每个客户端上发起请求。

相对于中心化的负载均衡器，Ribbon作为客户端的负载均衡器会有很多额外的收益。因为有一个在客户端本地的负载均衡器，所以负载均

衡器能够很自然地按照客户端的数量成比例伸缩。此外，每个负载均衡器都可以配置成最适合对应客户端的负载平衡算法，而不必对所有的服务都使用相同的配置。

如果你觉得它看上去有些复杂，那么不用担心，随后我们就会看到大多数功能都会以自动化、透明的方式来进行处理。在注册和消费服务之前，我们需要先启用Eureka服务器。

要开始使用Spring Cloud和Eureka，我们需要首先为Eureka本身创建一个全新的项目。最简单的方式是使用Spring Initializr，该项目可以使用任何名称，但是我一般会将其称为service-registry。在选择starter依赖的时候，我们只需要一项依赖：带有Eureka Server标签的复选框。在创建完新项目之后，在Initializr为我们生成的项目中，pom.xml将会包含如下依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

在pom.xml文件中，我们还可以看到名为spring-cloud.version的属性以及一个<dependencyManagement>区域，它们指定了Spring Cloud的发布版本。当我创建service-registry的时候，它引用的是Finchley train的第一个服务发布版本（SR1）：

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>
```

```
...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

如果你想要使用不同版本的Spring Cloud，只需要将spring-cloud.version属性修改为想要的版本即可。

在构建文件中添加完Eureka starter依赖之后，要启用Eureka服务器，我们还需要做一件事情，那就是打开应用的主引导类并为其添加@EnableEurekaServer注解：

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
  public static void main(String[] args) {
    SpringApplication.run(ServiceRegistryApplication.class, args);
  }
}
```

好的，这样就可以了！如果此时启动应用，Eureka服务注册中心就会运行起来并监听8080端口。如果此时在浏览器上访问http://localhost:8080，将会看到如图13.2所示的Web界面。

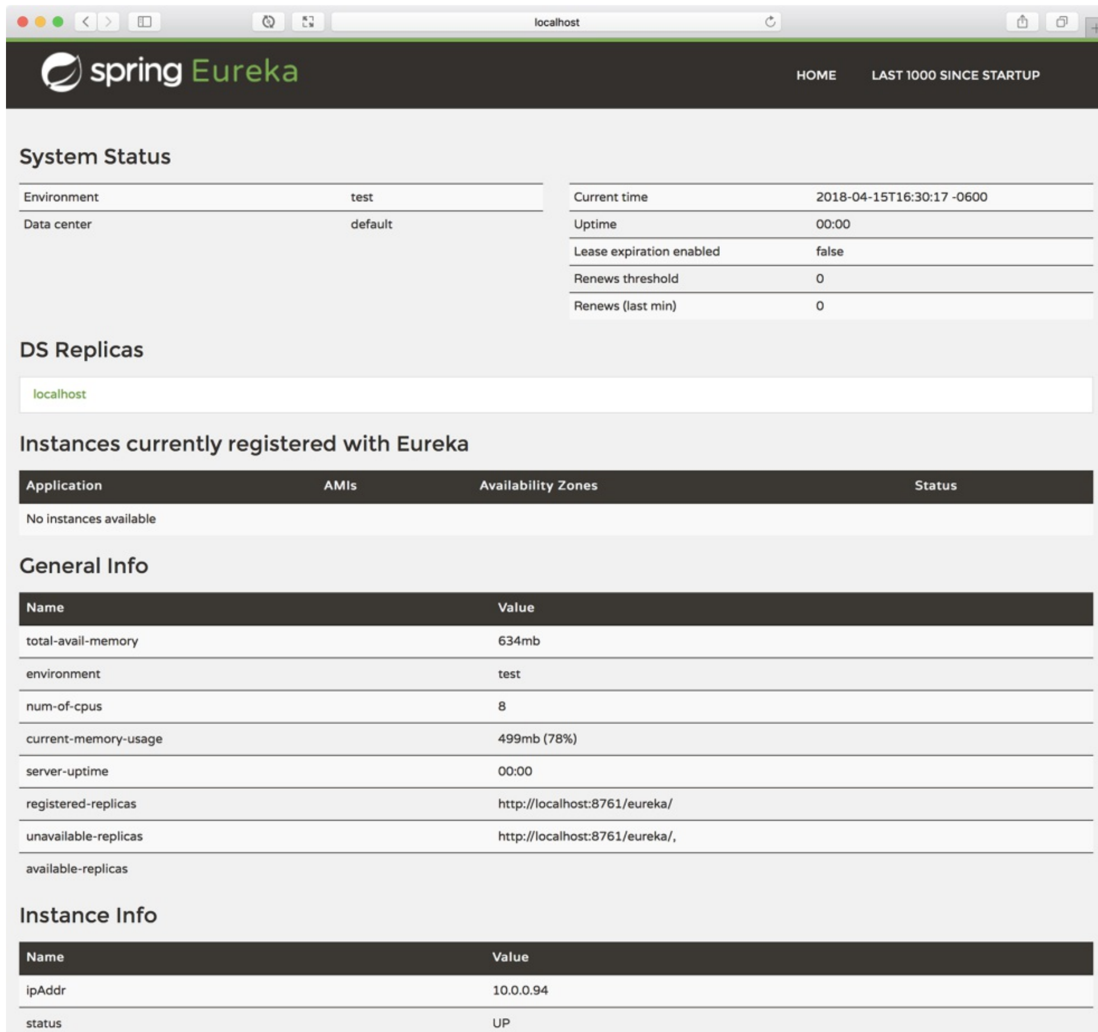


图13.2　Eureka基于Web的dashboard

Eureka的dashboard提供了丰富的信息，它会告诉我们都有哪些服务实例注册在Eureka上（当然还会有其他的信息）。在注册服务的时候，我们要经常访问这个UI界面，确保它们按照预期注册了进来。此时，还没有服务注册，所以提示信息是“No Instances Available”。

Eureka还对外暴露了REST API，借助它们服务可以自行进行注册，也可以发现其他的服务。你可能不会直接使用REST API，但是你会发现“/eureka/apps”端点非常有意思。它会列出注册中心所有服务实例的细

节。此时，我们没有注册任何服务，它的响应如下所示。在注册完服务之后，我们还会研究这个端点：

```
<applications>
  <versions__delta>1</versions__delta>
  <apps__hashcode></apps__hashcode>
</applications>
```

你会发现，在Eureka的日志中，每隔大约30秒就会打印出一些异常。不用担心，Eureka正在运行，而且完全符合我们的预期。但是，这些异常表明我们还没有完全配置好服务注册中心。接下来，我们添加一些配置属性来消除这些异常。

13.2.1 配置Eureka

Eureka不喜欢独自工作，并相信数量多会更安全的理念，希望能够成为Eureka服务器集群的一部分。如果有多个Eureka服务器，其中有一个遇到问题，就不会出现单点故障。因此，Eureka的默认行为是与其他Eureka服务器建立关联，尝试获取其他Eureka服务器的服务注册中心，甚至还会将自身注册为其他Eureka服务器的服务。

在生产环境中，Eureka的高可用是非常有价值的。但是，对于开发阶段来说，启动多个Eureka服务器既不方便也没有必要。为了达到开发的目的，有一个单独的Eureka服务器就足够了。除非我们正确配置了Eureka服务器，否则它会以日志文件中异常的形式每隔30秒就抱怨孤独状态。这是因为，每隔30秒，Eureka服务器就会尝试与另外的Eureka服务器建立关联，以注册自己并共享其注册中心中的信息。

我们需要做的就是配置Eureka使其接受当前的孤独状态。为了实现这一点，我们需要在application.yml中设置一些属性，代码片段如下所示：

```
eureka:
  instance:
    hostname: localhost
  client:
    fetch-registry: false
    register-with-eureka: false
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka
```

首先，我们将eureka.instance.hostname属性设置为localhost。这会告诉Eureka它正运行在哪个主机（host）上。这个属性是可选的，如果我们不指定它，那么Eureka会尝试通过环境变量确定它的主机。明确设置这个属性能够让我们更加确定它的值。

接下来的两个属性是eureka.client.fetch-registry和eureka.client.register-with-eureka。在其他的微服务中，我们可能会通过这两个属性告诉它们该如何与Eureka服务器进行交互。但是，不要忘了，Eureka也是一个微服务，所以这些属性也可以用到Eureka服务器上，以便于告诉它该如何与其他Eureka服务器进行交互。

这两个属性的默认值都是true，表明Eureka应该从其他的Eureka实例获取注册信息，并且应该将自身注册为其他Eureka服务器中的服务。因为在开发模式下并没有其他的Eureka服务器，所以我们将它们设置为false，这样Eureka将不会尝试与其他的Eureka服务器建立关联。

最后，我们还设置了eureka.client.service-url属性。这个属性包含了zone名称与该zone下一个或多个Eureka服务器之间的映射关系。defaultZone是一个特殊的key，如果客户端（在本例中，也就是Eureka本身）没有指定所需的zone，就将会使用这个zone。因为我们只有一个Eureka，映射到默认zone的URL就是Eureka服务器本身，所以这里使用了占位符变量，由其他属性填充它的值。

指定**Eureka**的服务器端口

尽管不一定是强制要求，但是我们可能想要修改默认的服务器端口。虽然Eureka非常乐意监听8080端口，但是在开发代码的时候我们可能会在本地机器同时运行多个应用（微服务），也就无法让所有的应用均监听8080端口。因此，在本地开发的时候，设置server.port属性通常是一个比较好的做法：

```
server:
  port: 8761
```

在这里，我们将端口设置成了8761，这是Eureka客户端（我们将会在13.3节中进行讨论）默认监听的端口。

禁用自我保护模式

另外一个我们需要考虑设置的属性是eureka.server.enable-self-preservation。如果我们启动Eureka服务器并让它空闲一分钟以上，可能就会在Eureka UI上看到一个非常吓人的错误信息，如图13.3所示。

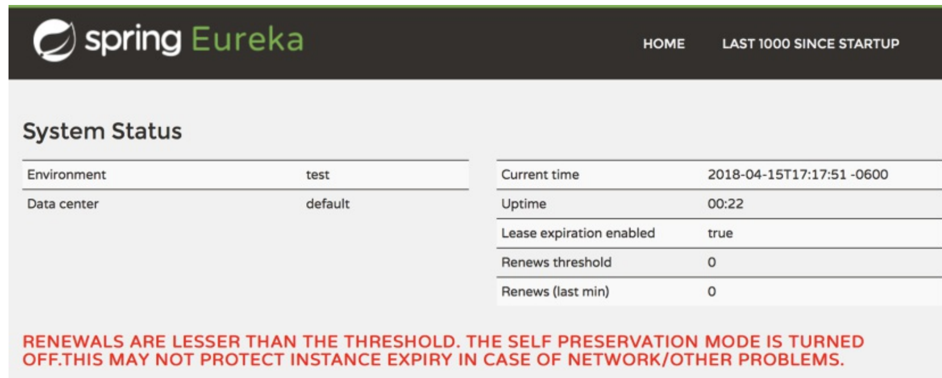


图13.3　在自我保护模式下，Eureka会在dashboard显示信息

尽管这里使用了红色字体和大写字母，但是这条信息并不像看上去那么严重。Eureka希望服务实例能够注册上来，并且每隔30秒向它发送一次注册更新的请求。通常，如果Eureka在3个更新周期（或者说90秒）内没有收到服务的更新请求，就会将该服务注销。在本例中，Eureka假定出现了网络问题，进入自我保护模式，所以不会注销服务实例。

在生产环境中，自我保护模式是很好的，可以防止在出现网络故障时更新请求无法发送至Eureka所导致的活跃服务被注销。但是，在我们第一次启动Eureka并且还没有注册任何服务时候，出现这样的告警会让人产生疑虑。我们可以将eureka.server.enable-self- preservation属性设置为false，从而禁用自我保护模式：

```
eureka:
  ...
  server:
    enable-self-preservation: false
```

这个属性在开发环境中是非常有用的。在开发环境中，基于各种原

因，Eureka可能会收不到更新请求。在这种环境下，我们可能会频繁地启动或关闭服务实例，自我保护模式会将已停止服务的注册项保留下来，另一个服务访问已经不可用的服务时就会产生问题。禁用自动保护模式将会防止这种诡异的问题。然而，我们付出的代价就是会看到另一条恐怖的红色信息（见图13.4）。

THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.

图13.4　禁用自我保护模式时，提示自我保护模式已禁用

虽然我们在开发环境可以禁用自我保护模式，但是在投入生产环境时需要将其启用。

13.2.2　扩展Eureka

在开发环境中，单个Eureka实例会更加便利；但是在将应用投入生产环境时，我们可能至少需要两个Eureka实例，以实现高可用性。

生产环境可用的**Spring Cloud Services**

在将微服务部署到生产环境时，有许多需要考虑的因素。Eureka的高可用性和安全性在开发阶段可能并不太重要，但是在生产环境中就非常关键了。如果你是Pivotal Cloud Foundry或Pivotal Web Services的客户，就可以让他们来关心这些事情了。

Spring Cloud Services提供了一个Eureka实现，同时还包含了配置服务器和断路器dashboard。我们所需要做的就是从marketplace请求一个p-

service-registry服务，然后将自己的微服务绑定到该服务上。在marketplace中，配置服务器和断路器dashboard（我们将会在接下来的两章中讨论它们）的名称分别为p-config-server和p-circuit-breaker-dashboard。

配置两个（或更多）Eureka实例最简单直接的方式就是在application.yml中使用Spring profile，然后针对两个profile各启动一次。例如，程序清单13.1中的配置项会将两个Eureka服务器设置为彼此对等的端。

程序清单13.1　使用Spring profile将Eureka配置成两个对等的端

```
eureka:
  client:
    service-url:
      defaultZone: http://${other.eureka.host}:${other.eureka.port}/eureka

---
spring:
  profiles: eureka-1
  application:
    name: eureka-1

server:
  port: 8761

eureka:
  instance:
    hostname: eureka1.tacocloud.com

other:
  eureka:
    host: eureka2.tacocloud.com
    port: 8761

---
spring:
  profiles: eureka-2
```

```
  application:
    name: eureka-2

server:
  port: 8762

eureka:
  instance:
    hostname: eureka2.tacocloud.com

other:
  eureka:
    host: eureka1.tacocloud.com
    port: 8762
```

在默认的profile中（位于程序清单13.1顶部），我们用占位符变量来设置eureka.client. service-url.defaultZone属性，这些占位符都是在每个profile特定的配置中设置的。

在默认的profile之后，我们配置了两个profile，分别为eureka-1和eureka-2。每个profile都按照自己的配置需要指定了端口和eureka.instance.hostname。随后，我们设置了两个略显牵强的other.eureka.host和other.eureka.port属性，在每个profile中它们都指向了其他的Eureka实例。这两个属性与框架本身是没有关系的，但是在默认profile的占位符中会引用它们。

注意，我们在这里没有设置eureka.client.fetch-registry或eureka.client.register-with-eureka。它们的默认值为true，因此能够确保每个Eureka服务器都会向对方进行注册，并且能够从其他Eureka服务器上获取注册信息。

目前，Eureka服务注册中心已经启动并处于运行状态了。但是，它

现在就像一个没有人查阅的空电话本。只有让服务开始在注册中心注册，并让其他服务查找和调用它们才行，否则我们的工作都是徒劳的。接下来，我们看一下如何让微服务成为Eureka的客户端。

13.3 注册和发现服务

没有服务注册的话，Eureka服务注册中心没有任何用处。如果你的服务想要被其他服务发现和消费，就需要将它们作为服务注册中心的客户端。为了让应用（任何应用，但很可能是微服务）成为服务注册中心的客户端，我们至少需要将Eureka客户端依赖添加到服务应用的构建文件中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

与Eureka服务器starter依赖类似，我们还需要为Spring Cloud的依赖管理设置Spring Cloud的版本属性：

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
```

```
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

我们可以手动添加这些条目到服务应用的pom.xml文件中，但是更简单的方式是在Spring Initializr的复选框中选中Eureka Discovery依赖。

Eureka client starter依赖会添加通过Eureka发现服务所需的所有内容，包括Eureka的客户端库以及Ribbon负载均衡器。我们只需要将这个依赖添加进来，就能将应用变成Eureka服务注册中心的客户端。当应用启动的时候，它会尝试联系在本地运行并且端口为8761的Eureka服务器，并将自身基于UNKNOWN名称进行注册。

13.3.1 配置Eureka客户端属性

对于开发阶段来说，默认位置的Eureka服务器是可以接受的，如果我们要将服务部署到localhost之外，就需要覆盖它的值。另外，默认的服务名为UNKNOWN，这是一个非常糟糕的选择……但是，坦白来讲，任何形式的默认方案都会很糟糕，因为如果采用默认方案，那么所有服务都会具有相同的名称。

更改服务在Eureka中的注册名称非常简单，我们只需要设置spring.application.name属性就可以了。例如，如果想要注册一个处理taco配料相关操作的服务，那么我们可以将其注册为ingredient-service。在application.yml中，将会如下所示：

```
spring:
  application:
    name: ingredient-service
```

设置完这个属性之后，我们就可以按照ingredient-service名称来查找服务了。另外，如果我们为配料服务添加多个实例，它们就会以相同的名称出现在注册中心，实际上，服务会扩展到多个实例，并假定它们是完全相同的，服务的消费者可以从中选择。此时，我们查看Eureka dashboard的话，服务将会如图13.5所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
INGREDIENT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.3:ingredient-service

图13.5　Eureka dashboard上的配料服务

在继续使用Spring Cloud的过程中，你会发现spring.application.name是我们要设置的最重要的属性之一。它决定了Eureka中的注册名。在第14章中我们将会看到，这个属性会帮助配置服务识别该应用，用来管理特定应用的配置。其他的Spring Cloud项目，如Spring Cloud Task（短暂存活的微服务）和Spring Cloud Sleuth（分布式跟踪），同样依赖spring.application.name属性来识别服务。

正如我们在第1章所学到的，默认情况下，所有的Spring MVC和Spring WebFlux应用都会监听8080端口。因为这些服务现在只会通过Eureka进行查找，所以它们监听什么端口也就无所谓了，Eureka能够知道它们使用的是什么端口。为了避免本地运行时潜在的端口冲突，我们可以将端口设置为0：

```
server:
  port: 0
```

注意：将端口设置成0的话，应用会选择任意一个可用端口来启动。

现在，我们要考虑Eureka服务器的位置。默认情况下，Eureka客户端会假定Eureka服务器在本地运行（8761端口）。对于开发期来说，这种方式很不错，但是在生产环境中，大多数情况并非如此。因此，我们需要指定Eureka服务器的位置。这与Eureka服务器本身的实现方式完全相同，都是要使用eureka.client.service-url属性：

```
eureka:
 client:
   service-url:
     defaultZone: http://eureka1.tacocloud.com:8761/eureka/
```

通过这样的配置，客户端会使用eureka1.tacocloud.com主机（端口8761）上的Eureka服务器进行注册。只要Eureka服务器在运行，这种方式就是没有问题的，但是一旦Eureka服务器因为某种原因而停机，服务注册就会失败。为了避免注册失败，最好是为服务配置两个或更多的Eureka实例：

```
eureka:
 client:
   service-url:
     defaultZone: http://eureka1.tacocloud.com:8761/eureka/,
                  http://eureka2.tacocloud.com:8762/eureka/
```

当服务启动的时候，它会尝试使用zone中的第一个服务器进行注册。如果因为某种原因失败，它将会使用列表中的下一个服务器来进行注册。最终，如果出现故障的服务器重新恢复在线状态，它将会从对等的端上复制注册信息，这样就能将该服务的注册条目包含进来。

在Eureka中注册服务只完成整个任务的一半。服务在Eureka注册之后，其他的服务就可以发现和消费它们了。接下来，我们看一下如何消费Eureka中注册的服务。

13.3.2　消费服务

在消费者代码中，硬编码任何服务实例的URL都是错误的做法。这不仅会让消费者与服务的特定实例耦合在一起，而且一旦服务的主机和/或端口改变，消费者就会出问题。

对于消费者应用来说，在从Eureka中查找服务时，它要承担很多责任。Eureka可能会基于查找结果返回同一个服务的多个实例。如果消费者请求ingredient-service服务时得到了多个服务实例，那么它该如何选择正确的服务呢？

好消息是消费者应用根本不需要从中进行选择，甚至都不需要自己显式地进行服务查找。借助Spring Cloud的Eureka客户端支持和Ribbon客户端负载均衡器，我们可以很容易地查找、选择和消费服务实例。我们有两种方式可以消费从Eureka中查找到的服务：

- 支持负载均衡的RestTemplate；
- Feign生成的客户端接口。

选择哪种方式在很大程度上取决于个人喜好。下面我们将会看一下这两种方案（首先会介绍支持负载均衡的RestTemplate），然后你就可以从中选择最喜欢的方式了。

使用**RestTemplate**消费服务

你对Spring RestTemplate客户端的第一印象可能来源于第7章。我们快速回忆一下它的运行原理，在创建或注入RestTemplate之后，我们就可以发送HTTP调用并将响应绑定到领域类型上。例如，为了发送根据ID获取配料的HTTP GET请求，我们可以使用如下的RestTemplate代码：

```
public Ingredient getIngredientById(String ingredientId) {
  return rest.getForObject("http://localhost:8080/ingredients/{id}",
                           Ingredient.class, ingredientId);
}
```

在这里，唯一的问题在于getForObject()的URL硬编码了特定的主机和端口。我想，你可能会将细节信息提取到一个属性中，但是如果我们将请求的目的地设置成配料服务众多实例中的某一个，那么我们所配置的URL会始终都指向同一个特定实例，这样就没有负载均衡器将请求分散到多个服务实例中了。

如果我们将应用变成Eureka客户端，就可以声明支持负载均衡的RestTemplate bean了。我们需要做的就是声明一个常规的RestTemplate

bean，并为带有@Bean注解的方法再添加上@LoadBalanced：

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

@LoadBalanced注解有两个目的。首先，也是最重要的，它会告诉Spring Cloud，这个RestTemplate要能够通过Ribbon来查找服务。其次，它会作为一个注入限定符（qualifier），所以有两个或更多RestTemplate bean的话，我们可以在注入的地方声明此处想要支持负载均衡的RestTemplate。

例如，就像上面的代码那样，我们想要使用支持负载均衡的RestTemplate来查找配料。首先，我们将支持负载均衡的RestTemplate注入需要它的bean中：

```
@Component
public class IngredientServiceClient {

  private RestTemplate rest;

  public IngredientServiceClient(@LoadBalanced RestTemplate rest) {
    this.rest = rest;
  }

  ...

}
```

随后，我们稍微修改一下getIngredientById()方法，使用服务的注册名，而不再明确使用主机和端口：

```
public Ingredient getIngredientById(String ingredientId) {
  return rest.getForObject(
              "http://ingredient-service/ingredients/{id}",
              Ingredient.class, ingredientId);
}
```

发现区别了吗？getForObject()的URL不再使用特定的主机名或端口。在主机名和端口的位置上，我们使用了服务名ingredient-service。在内部, RestTemplate会要求Ribbon根据名称查找服务并从中选择一个实例。Ribbon非常乐于效力，它会将URL重写为选定服务实例的主机和端口，然后让RestTemplate像以往那样进行处理。

我们可以看到，使用支持负载均衡的RestTemplate与标准RestTemplate并没有太大的差异。关键的不同点在于客户端需要使用服务名，而不是显式的主机名和端口。如果你想使用WebClient来替代RestTemplate该怎么办呢？WebClient也能够和Ribbon组合使用根据名称来消费服务吗？

使用**WebClient**消费服务

在第11章中，我们看到WebClient提供了与RestTemplate类似的HTTP客户端，但是它使用的是像Flux和Mono这样的反应式类型。如果你曾经被反应式编程的bug所困扰，那么你可能倾向于直接使用WebClient，而不是使用RestTemplate。好消息是，我们可以按照与RestTemplate类似的方式将WebClient作为支持负载均衡的客户端。我们需要做的第一件事就是声明一个返回WebClient.Builder bean的方法，该方法要添加@LoadBalanced注解：

```
@Bean
@LoadBalanced
public WebClient.Builder webClientBuilder() {
  return WebClient.builder();
}
```

在声明完WebClient.Builder之后，我们就可以将支持负载均衡的WebClient.Builder注入任何需要它的地方。例如，我们可以将它注入IngredientServiceClient的构造器中：

```
@Component
public class IngredientServiceClient {

  private WebClient.Builder wcBuilder;

  public IngredientServiceClient(
        @LoadBalanced WebClient.Builder webclientBuilder wcBuilder) {
    this.wcBuilder = wcBuilder;
  }

  ...

}
```

最后，在我们需要使用它的时候，可以利用WebClient.Builder构建一个WebClient，然后就能够使用Eureka注册的服务名来发送请求了：

```
public Mono<Ingredient> getIngredientById(String ingredientId) {
  return wcBuilder.build()
    .get()
      .uri("http://ingredient-service/ingredients/{id}", ingredientId)
    .retrieve().bodyToMono(Ingredient.class);
}
```

与支持负载均衡的RestTemplate类似，在发送请求的时候，这里不需要明确指定主机和端口。系统会从给定的URL中抽取出服务名，通过这个名称在Eureka中查询服务。Ribbon会选择服务的一个实例，在真正

发送请求之前，会根据所选实例的主机和端口重写URL。

这种编程模型非常容易掌握，若你已经熟悉RestTemplate或WebClient则更是如此。Spring Cloud还有一个技巧，接下来我们看一下如何使用Feign创建基于接口的服务客户端。

定义**Feign**客户端接口

Feign是一个REST客户端库，使用一种特殊的、接口驱动的方式来定义REST客户端。简而言之，如果你喜欢Spring Data自动实现repository接口的方式，那么你肯定会喜欢Feign的。

Feign最初是Netflix的一个项目，后来变成了独立的开源项目，名为OpenFeign。单词feign的意思是“伪装”，稍后我们将会看到对于假装成REST客户端的项目，这是一个很合适的名称。

要使用Feign，我们首先需要将依赖添加到项目的构建文件中。在pom.xml文件中，如下的<dependency>就可以完成该任务：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

在使用Spring Initializr的时候，我们可以通过选中Feign复选框自动添加该starter依赖。令人遗憾的是，目前不会根据已有的依赖启用自动配置功能。所以，我们需要将@EnableFeignClients添加到某个配置类上：

```
@Configuration
@EnableFeignClients
public RestClientConfiguration {
}
```

现在，到了有意思的部分。假设我们想要通过注册在Eureka中名为ingredient-service的服务获取一个Ingredient，需要做的就是定义如下的接口：

```
package tacos.ingredientclient.feign;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import tacos.ingredientclient.Ingredient;

@FeignClient("ingredient-service")
public interface IngredientClient {
  @GetMapping("/ingredients/{id}")
  Ingredient getIngredient(@PathVariable("id") String id);
}
```

这是一个很简单的接口，并没有实现类。在运行期，当Feign发现它的时候，这一切就都不重要了，Feign会自动创建一个实现类并将其暴露为Spring应用上下文中的bean。

仔细观察一下，我们会发现其中有一些注解在发挥作用，并将所有功能组合在了一起。接口上的@FeignClient注解会指定该接口上的所有方法都会对名为ingredient-service的服务发送请求。在内部，服务将会通过Ribbon进行查找，这与支持负载均衡的RestTemplate运行方式是一样的。

随后就是getIngredient()方法，它使用了@GetMapping注解。你会发现，这个注解来源于Spring MVC。确实，就是同一个注解。现在它用在

了客户端，而不是用在控制器上。它表明，任何对getIngredient()的调用都会对“/ingredients/{id}”路径发起GET请求，其中的主机和端口是通过Ribbon选定的。@PathVariable注解同样来自Spring MVC，会将方法参数映射到给定路径的占位符上。

现在，我们需要做的就是将Feign实现的接口注入需要的地方并开始使用它。例如，要在控制器中使用它，我们可以这样做：

```
@Controller
@RequestMapping("/ingredients")
public class IngredientController {

  private IngredientClient client;

  @Autowired
  public IngredientController(IngredientClient client) {
    this.client = client;
  }

  @GetMapping("/{id}")
  public String ingredientDetailPage(@PathVariable("id") String id,
                                     Model model) {
    model.addAttribute("ingredient", client.getIngredient(id));
    return "ingredientDetail";
  }
}
```

我不知道你的观点如何，但是我觉得这非常流畅！很难说我最喜欢哪种方式：支持负载均衡的RestTemplate、WebClient，还是具有魔力的Feign客户端接口。不管选择哪种方式，我们的REST客户端都能根据名称消费在Eureka注册的服务，避免硬编码特定的主机名和端口。

值得一提的是，Feign提供了自己的注解。@RequestLine和@Param非常类似于Spring MVC中的@RequestMapping和@PathVariable，但是它

们的使用方式略有差异。能够在客户端使用我们已经非常熟悉的Spring MVC注解是非常棒的，而且它们很可能与我们在定义服务控制器时所使用的注解是一样的。

13.4 小结

- 借助自动配置和@EnableEurekaServer 注解，Spring Cloud Netflix能够让我们很容易地创建Netflix Eureka服务注册中心。
- 微服务可以使用名字将它们自身注册到Eureka中，这样可以被其他服务发现。
- 在客户端，作为客户端负载均衡器，Ribbon能够根据名称查找服务并选择实例。
- 客户端代码可以使用RestTemplate，利用Ribbon进行负载均衡；也可以将REST客户端定义为接口，由Feign在运行期自动实现。
- 不管采用哪种方案，客户端代码都不需要硬编码它们所消费的服务的地址。

---

[1]　是的，在这部电影中，真正的问题是喂Mogwai的时间，也就是午夜时分。没有一个类比是完美的。

[2]　在这里，我们会关注如何使用Java和Spring编写微服务。如果你对如何使用.NET编写微服务并与Spring Cloud服务交互感兴趣，那么可以参考一下Steeltoe。

第14章　管理配置

本章内容：

- 运行Spring Cloud Config Server
- 创建Config Server的客户端
- 存储敏感配置
- 自动化刷新配置

买过房子或汽车的人可能都会面临厚厚的一叠纸。购买大宗商品时要签署的合同往往会对无纸化社会的承诺不屑一顾。每当我与汽车经销商或代理人坐到一起的时候，都感觉我应该提前准备好一叠绷带，为这个过程中几乎总能出现的纸划伤手的情况做好准备。

近年来，尽管我必须要签署的总页数几乎没有什么变化，但是我不必像以前那样填写那么多的字段了。对于表格中那些曾经手动填写的地方，现代化的表格在打印之前通常就基于收集到的数据预先填充好了。这样的话，不但会加快处理速度，而且能够减少在多个表格间手动填写

重复数据所导致的错误。

与之类似，很多应用程序都存在某种形式的配置。在第5章中，我们讨论了通过配置属性来设置Spring Boot应用。通常，我们设置的属性是该应用特有的，所以可以通过application.properties或application.yml文件声明这些属性，并将它们打包到应用的部署文件中。

按照微服务的方式来组织架构的话，多个服务之间的配置属性是通用的。就像手工填写带有重复数据的表单非常乏味而且易于出错一样，跨多个应用服务重复进行配置可能也会存在问题。

在本章中，我们将会研究Spring Cloud的Config Server，这是为指定应用中所有服务提供集中式配置的一个服务。借助配置服务器，我们可以在一个地方管理所有的应用配置，避免任何重复。

但是在开始之前，我们简单思考一下单独配置微服务的问题，以及中心化的配置为何能够更好。

14.1　共享配置

就像我们在第5章所看到的那样，我们可以通过多种属性源设置属性来对Spring应用进行配置。如果某个配置属性可能会更改或者只针对运行时环境有效，那么Java系统属性或操作系统环境变量是一个合适的可选方案。对于不太可能发生变化或者应用特定的属性，将它们放到application.yml或application.properties中，随着打包的应用一起部署是一种很好的方案。

这些方案对于简单的应用来说都很不错。但是，当在环境变量或Java系统属性中设置配置属性的时候，我们必须要接受这样一个现实，那就是修改这些属性需要应用重启。如果我们选择将属性打包到要部署的JAR或WAR文件中，那么在属性变更时，我们必须要完全重新构建和重新部署应用。如果我们想要回滚配置变更，那么同样的约束依然有效。

这些约束在有些应用程序中是可以接受的。但是，在有些情况下，如果仅仅是为了修改一个属性就重启应用，往好了说是不太方便，往坏了说则具有破坏性。除此之外，在基于微服务架构的应用中，属性管理会跨越多个代码库和部署实例，因此将相同变更用到应用中多个服务的每个实例中是不现实的。

有些属性是敏感的，比如数据库密码和其他类型的私密信息。尽管这些值作为应用的属性在写入的时候可以进行加密，但是应用在使用它们之前必须要先解密。即便如此，有些属性甚至可能需要对应用开发人员保密。这样的话，将它们设置成环境变量或者将它们与应用的其他代码一起通过源码控制系统进行管理就是不可取的了。

相反，我们可以考虑一下这些场景在集中式的配置管理下会是什么样子。

- 配置不再需要和应用程序代码一起打包和部署。这样的话，配置的变更或回滚就都不需要重新构建和重新部署应用了。配置甚至可以在运行时进行变更，无须重新启动应用。
- 共享通用配置的微服务不需要管理自己的属性设置副本，并且能够

管理共享的相同属性。如果需要对属性进行变更，那么这些变更只需在一个地方执行一次就可以应用到所有的微服务上。

- 敏感配置可以进行加密，并且能够与应用代码分开进行维护。应用可以按需获取未加密的值，而不需要应用程序提供解密信息相关的代码。

Spring Cloud Config Server提供了中心化的配置功能，应用中的所有微服务均可以依赖该服务器来获取配置。因为它是中心化的，所以是一个一站式的配置商店，所有的服务都可以使用它，另外它还能够为特定服务提供专门的配置。

使用Config Server的第一步就是创建并运行该服务器。

14.2 运行配置服务器

Spring Cloud Config Server为配置数据提供了中心化的数据源。与Eureka类似，我们可以将Config Server视为另一个微服务，在更大的应用中，它的角色就是为应用中的其他服务提供配置数据。

Config Server暴露了REST API，客户端（也就是其他的服务）可以通过它来消费配置属性。通过Config Server提供的配置来源于Config Server之外，通常来源于一个像Git这样的源码控制系统。图14.1阐述了它是如何运行的。

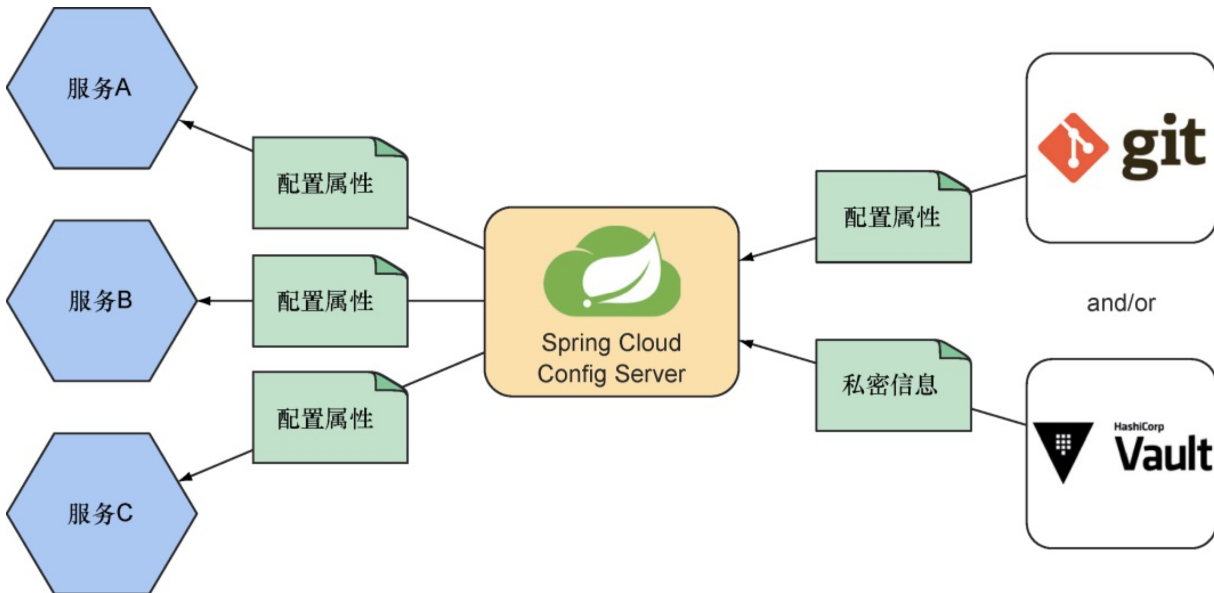


图14.1　Spring Cloud Config Server通过支撑的Git仓库或Vault私密存储来为其他服务提供配置属性

注意，在图14.1中，我使用的是Git的图标，而不是GitHub的图标。这是很重要的，我们可以使用任意的Git实现来存储配置信息，包括但不限于GitHub。GitLab、微软的Team Foundation Server和Gogs都是合法的Config Server后端可选方案。

> 注意：不管使用哪个Git服务器，Config Server几乎没有什么差异。在这里，我选择使用Gogs，这是一个轻量级、易于搭建的Git服务器。更具体来讲，我在开发使用的机器运行Gogs时完全遵循了Docker中运行Gogs的指南。

将配置信息存储在像Git这样的源码控制系统中，配置可以像应用源码那样实现版本化、使用分支、添加标签、恢复和指摘（blame）。

但是，为了让配置信息与使用它们的源码分离，这些配置可以独立于应用演化和版本化。

你可能注意到了，在图14.1中还包含了HashiCorp Vault。如果想要保持配置属性完全私密，并且要将它们锁起来直到需要的时候才取出，那么Vault非常有用。我们将会在14.5节中讨论如何组合使用Config Server和Vault。

14.2.1 启用配置服务器

作为更大应用系统中的一个微服务，Config Server会作为一个独立的应用进行开发和部署。所以，我们需要为Config Server创建一个全新的项目。要实现这一点，最简单的方式就是使用Spring Initializr或它的某个客户端（比如Spring Tool Suite中的New Spring Starter Project向导）。

配置：重载的术语

当我们讨论Spring Cloud Config Server的时候，会经常用到“配置（configuration）”这个术语，但是它所指的并不总是同一件事。我们将会编写配置属性来配置Config Server本身。同时，Config Server还会为应用提供配置属性。Config Server的名字中还有“Config”这个单词，这会导致一定的混乱。

在使用“configuration”这个单词的时候，我都会尽力表达清楚到底指的是哪个配置，而在代指Config Server的时候，我都会使用“Config”这个缩写形式。

我一般会将项目命名为“config-server”，但是你可以选取任何你喜欢的名称。最重要的是要选中Config Server复选框，这样就能声明对Config Server的依赖。这样做的结果就是会在所生成项目的pom.xml文件中添加如下的依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Config Server的版本是根据选择的Spring Cloud release train确定的。在pom.xml文件中，必须要配置Spring Cloud release train。在我编写本书的时候，最新的Spring Cloud发布版本是Finchley.SR1。所以，在pom.xml文件中将会发现如下的属性和<dependencyManagement>代码块：

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
```

```
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

尽管Config Server依赖将Spring Cloud添加到了项目的类路径下，但是这里并没有自动配置启动它，所以我们需要为某个配置类添加@EnableConfigServer。顾名思义，这个注解会在应用运行的时候启用一个Config Server。通常，我会将@EnableConfigServer放到主类中，如下所示：

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

在我们想要启动应用并查看Config Server如何运行之前，必须还要做另外一件事情：我们必须要告诉它，它要对外提供的配置属性都位于何处。作为开始，我们将会使用来自Git仓库的配置，所以我们需要将spring.cloud.config.server.git.uri属性设置为配置仓库的URL：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/tacocloud/tacocloud-config
```

在14.2.2小节，我们将会看到如何为Git仓库填充属性。

为了在本地开发环境运行，我们可能还要配置另一个属性。在测试本地服务的时候，我们最终会有多个服务一直运行并且它们要监听localhost的不同端口。作为典型的Spring Boot Web应用，Config Server默认会监听8080端口。为了避免端口冲突，我们可以通过设置server.port属性指定一个唯一的端口号：

```
server:
  port: 8888
```

在这里，我们将server.port设置为8888，是因为在14.3节中我们将会看到这是Config客户端试图获取配置信息时默认使用的端口。可以将其设置成任意值，但是在配置客户端服务中必须要与其匹配。

很重要的一点需要注意，我们此时所编写的配置是针对Config Server本身的。它与Config Server对外提供的配置是不同的。Config Server会对外提供从Git或Vault获取到的配置信息。

此时，如果启动应用，就会有一个监听8888端口的Config Server，它还不能提供任何的配置属性。我们目前还没有任何Config Server客户端，但是可以通过curl命令行（或者提供同样功能的HTTP客户端）模拟一个客户端：

```
$ curl localhost:8888/application/default
{
  "name": "application",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": "ca791b15df07ce41d30c24937eece4ec4b208f4d",
  "state": null,
```

```
  "propertySources": []
}
```

它会向Config Server的“/application/default”路径发送HTTP GET请求。这个请求可以由两部分或3部分组成，如图14.2所示。

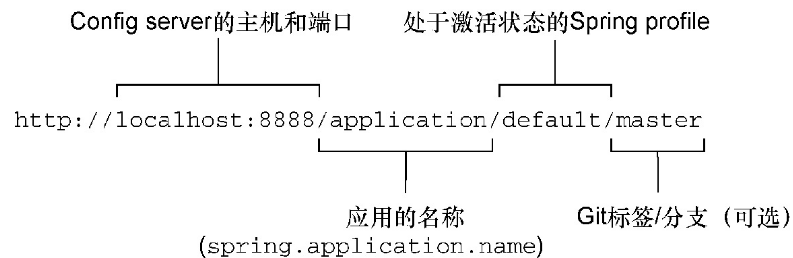


图14.2　Config Server对外暴露了一个REST API（通过它可以消费配置属性）

路径的第一部分，即“application”，指的是发送请求的应用的名称。在14.4.1小节中将会看到，Config Server是如何利用请求路径中这部分的内容为我们提供特定应用配置的。现在，我们没有特定应用的配置，所以任意值都是可以的。

路径的第二部分指的是发送请求的应用中处于激活状态的Spring profile。在14.4.2小节中，我们将会看到Config Server是如何利用请求路径中的profile值提供激活active特定配置的。我们目前没有特定profile的配置，所以任意的profile值都是可以的。

路径的第三部分是可选的，指定了提供配置信息的后端Git仓库的标签或分支。如果没有指定，那么默认会使用“master”分支。

请求的响应为我们提供了一些关于Config Server的基本信息，包括为我们提供配置信息的Git提交的版本和标签。但是，这里明显缺少的

就是真正的实际配置信息。正常情况下，我们会在propertySources属性下看到它们，但是在这个响应中，它是空的。这是因为我们需要为Git仓库填充Config Server要对外提供的属性。现在，我们看一下该如何实现。

14.2.2 填充配置仓库

我们有多种办法为Config Server提供属性，最基本、最直接的方案是提交application.properties或application.yml文件到Git仓库的根路径下。

假设我们已经推送了一个名为application.yml的文件到前面章节所配置的Git仓库下。这个配置文件与前面章节的配置是不同的，它是Config Server将要对外提供的配置。假设在这个application.yml文件中我们配置了如下的属性：

```
server:
  port: 0

eureka:
  client:
    service-url:
      defaultZone: http://eureka1:8761/eureka/
```

尽管这个application.yml文件的内容并不多，但是它所定义的配置是相当重要的。它会告诉应用中的每个服务都选择任意可用的端口并且告诉它们进行服务注册的Eureka在哪里。这意味着，在14.3节中，当我们将服务变成Config Server客户端的时候，我们可以从服务中移除显式

的Eureka配置。

作为Config Server的客户端，我们可以使用curl命令行查看Config Server提供的新配置数据：

```
$ curl localhost:8888/someapp/someconfig
{
  "name": "someapp",
  "profiles": [
    "someconfig"
  ],
  "label": null,
  "version": "95df0cbc3bca106199bd804b27a1de7c3ef5c35e",
  "state": null,
  "propertySources": [
    {
      "name": "http://localhost:10080/habuma/tacocloudconfig/
     application.yml",
      "source": {
        "server.port": 0,
        "eureka.client.service-url.defaultZone":
     "http://eureka1:8761/eureka/"
      }
    }
  ]
}
```

与之前对Config Server的请求不同，这个响应的propertySources属性中有了内容。具体来讲，它包含了一个属性源，属性源的name属性指向了Git仓库的引用，source则包含了我们推送至Git仓库中的属性。

从Git子路径下提供配置

按照代码的组织风格，你可能想要将配置信息存储到Git仓库的子目录下，而不是放到根路径下。例如，我们想要将配置放到相对于Git仓库根目录名为“config”的子目录下，就可以按照如下方式设置

spring.cloud.config.server.git.search-paths属性，让Config Server不再从根目录而是从“/config”目录下提供配置信息：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config
```

注意，spring.cloud.config.server.git.search-paths属性是一个复数形式，这意味着我们可以让Config Server提供来自多个路径的配置，只需将它们列出来以逗号分隔即可：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config,moreConfig
```

这样的话，Config Server会提供Git仓库下来自“/config”和“/moreConfig”路径的配置。

我们还可以使用通配符指定搜索路径：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config,more*
```

这里，Config Server会提供来自“/config”和所有以“more”开头的子目录的配置。

从**Git**分支或标签下提供配置

默认情况下，Config Server会提供Git中master分支下的配置。在客户端，我们可以将特定分支或标签设置为请求Config Server路径的第三个成员，如图14.2所示。但是，我们可能会发现让Config Server默认请求Git下特定的标签或分支会非常有用，而不是默认使用master。spring.cloud.config.server.git.default-label属性可以重写默认的标签或分支。

例如，考虑如下的配置，它会让Config Server提供名为“sidework”的分支（或标签）下的配置：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          default-label: sidework
```

按照这个配置形式，除非Config Server客户端指定，否则将会提供“sidework”分支下的配置。

为**Git**后端提供认证

Config Server检索配置信息的后端Git仓库很可能会使用用户名和密

码进行保护。如果是这样，我们就必须为Config Server提供Git仓库的凭证信息。

spring.cloud.config.server.username和spring.cloud.config.server.password属性可以为后端仓库设置用户名和密码。如下的Config Server配置将设置这些属性：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          username: tacocloud
          password: s3cr3tP455w0rd
```

在这里，分别将用户名和密码设置成了tacocloud和s3cr3tP455w0rd。

使用curl作为Config Server的客户端能够帮助我们体验一下Config Server是怎样运行的。实际上，Config Server所能做的远远不止于此。但是，我们所编写的微服务并不会使用curl来获取配置数据。所以在查看Config Server提供配置的其他方式之前，我们将关注点转移到微服务上，看一下如何将它们变成Config Server的客户端。

14.3　消费共享配置

除了提供中心化的配置服务器，Spring Cloud Config Server还提供了一个客户端库，它会包含在Spring Boot应用的构建文件中，允许应用

成为Config Server的客户端。

将Spring Boot应用变成Config Server客户端的最简单方式就是添加如下的依赖到项目的Maven构建文件中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

相同的依赖也可以在Spring Initializr中通过选择标签为Config Client的复选框添加进来。

当应用启动的时候，自动配置功能将会自动化地注册一个属性源，该属性源将会从Config Server中拉取属性。默认情况下，它会假定Config Server运行在localhost并监听8888端口。如果情况并非如此，我们可以通过设置spring.cloud.config.uri配置Config Server的位置：

```
spring:
  cloud:
    config:
      uri: http://config.tacocloud.com:8888
```

需要清楚一点，这些属性必须要放到Config Server客户端应用的本地，比如随每个微服务打包和部署的application.yml或application.properties文件中。

现在，我们有了一个中心化的配置服务器，几乎所有的配置都将会由它来提供，每个微服务都不需要携带很多自己的配置了。正常情况下，我们只需要设置spring.cloud.config.uri属性来指定配置服务器的地址

并设置spring.application.name属性为配置服务器指明当前应用即可。

哪个优先：**Config Server**还是服务注册中心？

我们正在设置微服务，让它们通过Config Server了解Eureka服务注册中心在什么地方。这是一种通用的方式，能够避免在应用的每个微服务中重复服务注册中心的细节信息。

同时，我们还可能会将Config Server本身注册到Eureka中，并让每个微服务像发现其他服务那样去查找Config Server。如果你喜欢这种模式，就需要将Config Server变成服务发现的客户端，并将spring.cloud.config.discovery.enabled属性设置为false。这样的话，Config Server会将自身以“configserver”名称注册到Eureka中。

这种方式的缺点在于，每个服务在启动的时候都要调用两次外部的服务：第一次调用Eureka发现Config Server的位置，第二次调用Config Server获取配置数据。

当应用启动的时候，Config Server客户端提供的属性源将会对Config Server发送请求。它所接收到的属性将会放到应用的环境之中。除此之外，这些属性实际上还会被缓存起来，即便Config Server停机，它们依然是可用的（我们将会在14.6节看一下在属性发生变更的时候，刷新它们的几种方式）。

到目前为止，Config Server提供的配置都非常简单，面向所有的应用和所有的profile。但有时候，我们需要提供特定应用专有的配置，或者提供当应用在特定profile处于激活状态时才可用的配置。我们看一下Config Server的另一面，看看使用它的几种方式，包括提供特定应用和特定profile的属性。

14.4 提供特定应用和profile的属性

我们可以回忆一下，当Config Server客户端启动的时候，它会发送一个请求到Config Server中，这个请求的路径中会包含应用的名称和激活profile的名称。在提供配置数据的时候，Config Server会考虑这些值，并为客户端返回特定应用和特定profile的配置数据。

从客户端的角度来讲，消费特定应用和特定profile的配置属性与之前没有Config Server时并没有太大的差别。应用的名称可以通过spring.application.name属性（这与Eureka识别应用的属性名是相同的）来指定应用的名称。激活的profile可以通过spring.profiles.active属性进行设置（通常会通过名为SPRING_PROFILES_ACTIVE的环境变量进行设置）。

类似的，要提供面向特定应用和profile的属性，Config Server本身也没有太多需要做的。真正比较重要的是，这些属性在支撑Git仓库中该如何进行存储。

14.4.1 提供特定应用的属性

按照我们之前的讨论，使用Config Server的好处之一就是我们可以让应用中的所有微服务共享通用的配置属性。尽管如此，有些属性可能是某个服务特有的，不需要（或者不应该）与所有的服务共享。

除了共享配置之外，Config Server还能管理面向特定应用的配置属性。要实现这一点，需要将配置文件的名称命名为该应用spring.application.name属性的值。

在第 13 章中，我们使用spring.application.name属性为微服务提供了一个名称，将会注册到Eureka中。相同的属性也可以被配置客户端用来在Config Server中识别自身，这样Config Server就能提供该应用特有的配置。

例如，在Taco Cloud应用中，我们将应用拆分成了多个微服务，分别是ingredient-service、order-service、taco-service和user-service，我们可以在每个服务的spring.application.name属性中指定它的名称。然后，我们就可以根据各个服务的名称在Config Server的Git后端创建对应的配置YAML文件，比如ingredient-service.yml、order-service.yml、taco-service.yml和user-service.yml。图14.3为 Gogs Web应用中配置仓库的文件截图。

不管服务应用的名称是什么，所有的应用都会接收来自application.yml文件的配置。但是，在向Config Server发起请求的时候，每个服务应用的spring.application.name的属性值会一同发送（作为请求路径的第一部分），如果存在匹配的属性文件，那么该文件中的属性将

会一并返回。如果application.yml中通用的属性与特定应用配置文件中的属性出现重复，那么特定应用的属性会优先生效。

需要注意的是，尽管图14.3显示的是YAML配置文件，实际上，如果在Git仓库中存放properties文件，同样的规则依然有效。

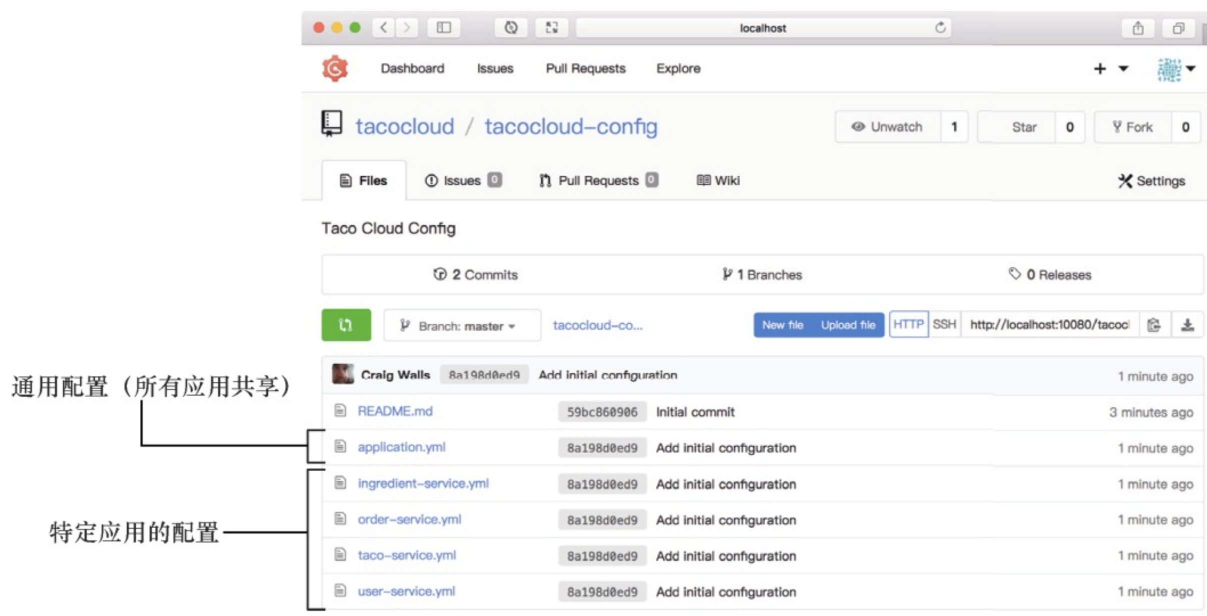


图14.3　应用特定的配置文件会根据每个应用的spring.application.name属性进行命名

14.4.2　提供来自profile的属性

在第5章中，在编写配置属性时，我们曾经看到过利用Spring profile实现特定的属性只有在给定profile处于激活状态时才生效。Spring Cloud Config Server采用与单个Spring Boot应用完全相同的方式，提供了对特定profile属性的支持，包括：

- 提供特定profile的“.properties”或YAML文件，比如名为application-

production.yml的配置文件；

- 在一个YAML文件中提供多个profile配置组，它们之间以“---”和spring.profiles分割开。

假设我们要通过Config Server为应用所有的微服务共享Eureka配置，现在它只引用了一个Eureka开发实例，对于开发环境来说是很不错的。如果服务要在生产环境运行，那么我们可能想要将它配置成引用多个Eureka节点。

另外，尽管我们在开发环境的配置中将server.port属性设置成了0，但是服务在部署到生产环境的时候，每个服务可能会运行到独立的容器中，容器将8080端口映射到外部的端口，这样就需要所有的应用都监听8080端口。

借助profile，我们可以声明多个配置。除了已经推送到Config Server Git后端的默认application.yml文件之外，我们还可以推送另外一个名为application-production.yml的YAML文件，如下所示：

```
server:
  port: 8080

eureka:
 client:
   service-url:
     defaultZone: http://eureka1:8761/eureka/,http://eureka2:8761/eureka/
```

在应用从Config Server获取配置信息的时候，Config Server会识别哪个profile处于激活状态（位于请求路径的第二部分）。如果活跃profile是production，那么两个属性集（application.yml和application-

production.yml）都将会返回，并且application- production.yml中的属性会优先于application.yml中的默认属性。图14.4为后端Git仓库的显示效果。

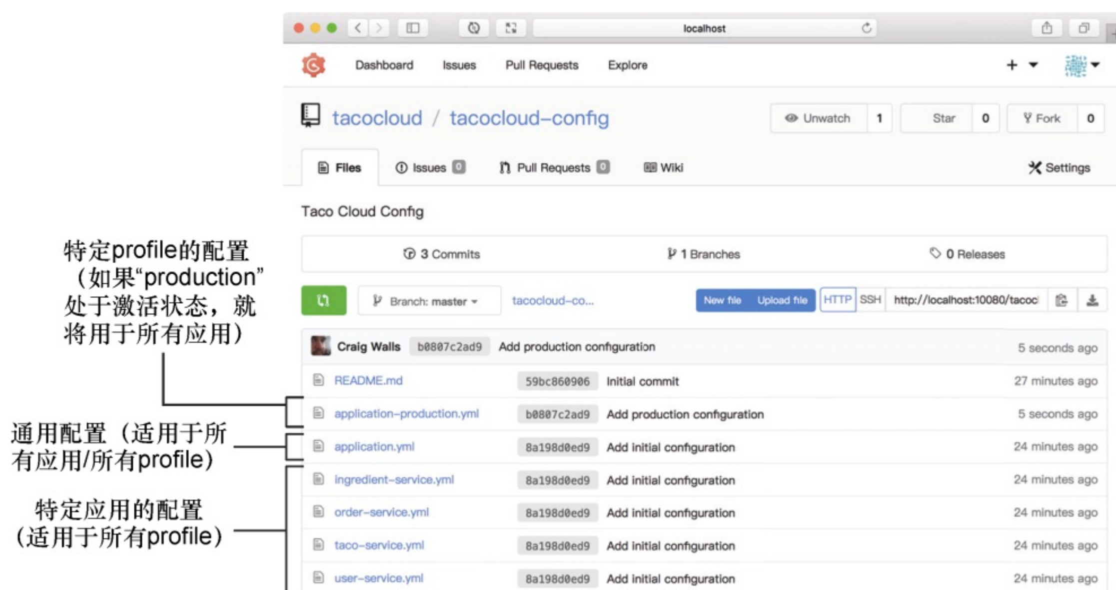


图14.4　特定profile的配置文件在命名时后缀与激活profile的名称相同

我们还可以使用同样的命名约定指定适用于特定应用且特定profile的属性，也就是将属性文件命名为应用名加中划线再加profile名的形式。

例如，我们想要为名为ingredient-service的应用设置属性，而且这些属性只有当production profile处于激活状态时才有效。在这种场景下，名为ingredient-service-production.yml的文件可以包含特定应用且特定profile的属性，如图14.5所示。

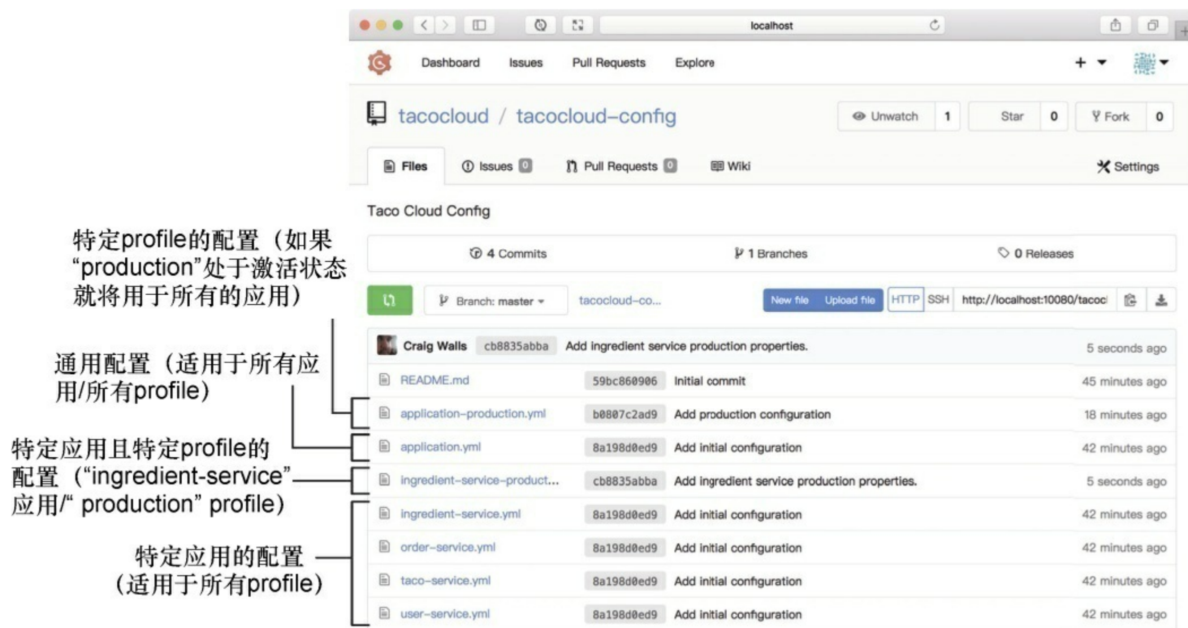


图14.5　配置文件可以适用于特定应用且特定profile的属性

对于特定profile的属性，在后端Git仓库中，我们也可以使用相同命名约定的properties文件来代替YAML。在YAML文件中，我们可以将特定profile的属性和默认profile的属性放到同一个文件中，中间使用3个中划线和spring.profiles进行分割，相关内容我们在第5章已经学习过了。

14.5　保持配置属性的私密性

Config Server提供的大多数配置可能并不是私密的。但是，我们可能需要Config Server提供一些包含敏感信息的属性，比如密码或安全token，在后端仓库中，它们最好保持私密。

Config Server提供了两种方式来支持私密的配置属性。

- 在Git存储的属性文件中使用加密后的值。

- 使用HashiCorp Vault 作为Config Server的后端存储，补充（或替代）Git。

我们将会依次看一下这两种方案是如何与Config Server组合使用保证配置属性私密性的。首先，我们看一下如何在Git后端中写入加密的属性。

14.5.1　在Git中加密属性

除了提供非加密值以外，Config Server也可以借助存储在Git中的属性文件提供加密值。处理存储在Git中的加密数据的关键在于一个秘钥（key），即加密秘钥（encryption key）。

为了启用加密属性功能，我们使用一个加密秘钥来配置Config Server，在将属性值提供给客户端应用之前，Config Server要使用这个秘钥对属性值进行解密。Config Server支持对称秘钥和非对称秘钥。要设置对称秘钥，我们可以在Config Server自己的配置中将encrypt.key属性设置为加密和解密秘钥的值：

```
encrypt:
  key: s3cr3t
```

很重要的一点需要注意，这个属性要设置到bootstrap配置中（例如，bootstrap.properties或bootstrap.yml）。这样的话，在自动配置功能启用Config Server之前，这个属性就会加载和启用。

为了更加安全一些，我们可以让Config Server使用非对称的RSA秘

钥对或引用一个keystore。要创建这样的秘钥，我们可以使用keytool命令行工具：

```
keytool -genkeypair -alias tacokey -keyalg RSA \
-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
-keypass s3cr3t -keystore keystore.jks -storepass l3tm31n
```

这样形成的keystore会写入到名为keystore.jks的文件中。我们可以将这个keystore.jks文件放到文件系统中或者放到应用本身。不管使用哪种方式，我们都需要在Config Server的bootstrap.yml文件中配置keystore的位置和凭证信息。

注意：为了在Config Server中使用加密功能，我们需要要安装Java Cryptography Extensions Unlimited Strength策略文件。参见Oracle的Java SE页面了解详细信息。

例如，假设我们要将keystore打包到应用本身，将其放到类路径的根目录下，那么我们可以配置如下的属性，让Config Server使用该keystore：

```
encrypt:
  key-store:
    alias: tacokey
    location: classpath:/keystore.jks
    password: l3tm31n
    secret: s3cr3t
```

秘钥和keystore就绪之后，我们需要对某些数据进行加密。Config

Server暴露了一个“/encrypt”端口会帮助我们实现该功能。我们需要做的就是提交一个POST请求到“/encrypt”端点，其中包括要加密的数据。例如，我们要加密连接至MongoDB数据库的密码。借助curl，我们可以按照如下的方式加密密码：

```
$ curl localhost:8888/encrypt -d "s3cr3tP455w0rd"
93912a660a7f3c04e811b5df9a3cf6e1f63850cdcd4aa092cf5a3f7e1662fab7
```

在提交POST请求之后，我们会接收到一个加密的值作为响应。接下来，需要做的就是复制这个值并粘贴到Git仓库托管的配置文件中。

为了设置MongoDB，在Git仓库的application.yml文件中添加spring.data.mongodb. password属性：

```
spring:
  data:
    mongodb:
      password: '{cipher}93912a660a7f3c04e811b5df9a3cf6e1f63850...'
```

需要注意，spring.data.mongodb.password被一个单括号（'）括了起来，并且带有{cipher}前缀。这样就会告诉Config Server，这是一个加密的值，而不是简单的未加密值。

在将这个变更提交并推送到Git仓库中的application.yml文件之后，Config Server就可以对外提供加密的属性了。如果要实际看一下，就使用curl命令伪装成Config Server的客户端：

```
$ curl localhost:8888/application/default | jq
{
  "name": "app",
  "profiles": [
```

```
    "prof"
  ],
  "label": null,
  "version": "464adfd43485182e4e0af08c2aaaa64d2f78c4cf",
  "state": null,
  "propertySources": [
    {
      "name": "http://localhost:10080/tacocloud/tacocloudconfig/
     application.yml",
      "source": {
        "spring.data.mongodb.password": "s3cr3tP455w0rd"
      }
    }
  ]
}
```

我们可以看到，spring.data.mongodb.password的值是以解密后的形式提供的。默认情况下，Config Server提供的所有加密值只是在后端Git仓库中处于加密的状态，它们在对外提供之前会解密。这意味着，消费这个配置的客户端应用并不需要任何特殊的代码和配置就能接收Git中已加密的属性。

如果你想要让Config Server以未解密的形式对外提供加密属性，那么可以将spring.cloud.config.server.encrypt.enabled属性设置为false：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
        encrypt:
          enabled: false
```

这样导致的结果就是Config Server在提供所有的属性值的时候完全按照Git仓库设置的样子进行发送，包括已加密的属性值。我们再次伪

装成一个客户端，利用curl命令展示禁用解密的效果：

```
$ curl localhost:8888/application/default | jq
{
  ...
  "propertySources": [
    {
      "name": "http://localhost:10080/tacocloud/tacocloudconfig/
     application.yml",
      "source": {
        "spring.data.mongodb.password": "{cipher}AQA4JeVhf2cRXW..."
      }
    }
  ]
}
```

当然，如果客户端接收到了未解密的属性值，那么客户端需要自行解密。

尽管可以通过Config Server在Git中保存已加密的私密信息，但是我们可以看到加密并不是Git的原生特性。它需要我们自己对写入Git仓库的数据进行加密。另外，除非将解密的任务推给Config Server的客户端应用，否则对于任何请求配置的客户端，Config Server API对外提供的私密信息都是解密之后的形式。我们接下来看一下另一个Config Server后端方案，它能够只向已授权的用户提供私密信息。

14.5.2　在Vault中存储私密信息

HashiCorp Vault是一个私密管理工具。这意味着与Git相比，Vault的核心特性就是原生地处理私密信息。对于敏感的配置数据，Vault是一个更有吸引力的Config Server后端支撑方案。

为了开始使用Vault，我们需要参考Vault Web站点的安装指南下载并安装vault命令行工具。在本小节中，我们将会使用vault命令管理私密信息和启动Vault服务器。

启动**Vault**服务器

在使用Config Server写入和对外提供私密信息之前，我们需要启动一个Vault服务器。对于我们来讲，最简单的方式就是在开发模式下使用如下的命令启动服务器：

```
$ vault server -dev -dev-root-token-id=roottoken
$ export VAULT_ADDR='http://127.0.0.1:8200'
$ vault status
```

第一条命令会在开发模式下启动一个Vault服务器，其中根token（root token）的ID为roottoken。顾名思义，开发模式意味着它是一个更简单但并不完全安全的Vault运行时。它不应该在生产环境中使用，但是在开发的工作流程中，这种使用Vault的方式会非常便利。

> 注意：Vault是一个功能完备且健壮的私密管理工具。除了开发模式下的简单使用之外，本章没有足够的篇幅完整介绍Vault服务器的运行。我强烈建议你在尝试生产环境中使用Vault之前，通过阅读Vaul文档来更详细地了解Vault。

对Vault服务器的所有访问都需要向服务器提供一个token。根token

是一个管理token，这意味着除了其他功能之外，它允许我们创建其他的token。它还能够用于读取和写入私密信息。如果在开发模式启动服务器的时候未指定根token，那么Vault服务器会为我们创建一个token并在启动的时候写入日志中。为了便于使用，建议将根token设置成一个易于记忆的值，比如roottoken。

开发模式的服务器启动之后，它将会监听本地机器的8200端口。所以，要让vault命令行知道Vault服务器在什么地方，设置VAULT_ADDR环境变量是非常重要的，这也是上述代码片段第二行所做的事情。

最后，vault status命令会校验之前的两条命令是否已经按照预期运行。你大致会看到描述Vault服务器的6个属性，包括Vault是否密闭（在开发模式下，它不应该处于密闭状态）。

使用Vault 0.10.0或之后的版本的话，Vault与Config Server协作使用之前还有其他的两条命令需要执行。Vault运行方式的一些变更会导致一个标准的私密后端与Config Server不兼容。以下两个命令会重新创建名为secret的后端，以兼容Config Server：

```
$ vault secrets disable secret
$ vault secrets enable -path=secret kv
```

如果使用更早版本的Vault，就不需要这些步骤。

写入私密信息到**Vault**中

借助vault命令，可以很容易将私密信息写入Vault中。例如，假设

我们想要将访问MongoDB的密码（也就是spring.data.mongodb.password）存储到Vault中，而不是存储到Git里面，就可以通过vault命令完成：

```
$ vault write secret/application spring.data.mongodb.password=s3cr3t
```

图14.6拆分了vault write命令，阐述每个组成部分在将私密信息写入Vault的过程中扮演了什么角色。

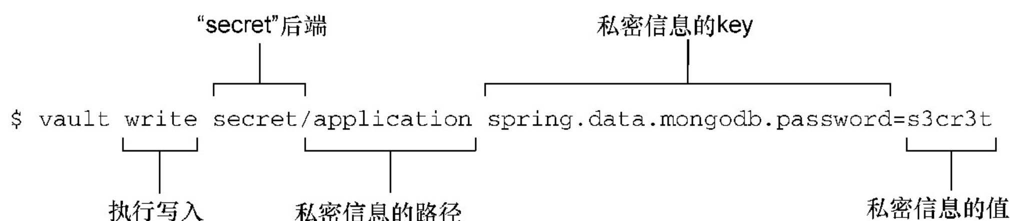


图14.6　通过vault命令将私密信息写入Vault

现在，我们最需要关注的就是私密信息的路径、key和值。私密信息的路径就像文件系统中的路径那样，允许我们将相关的私密信息放到一个给定的路径中，而将其他的私密信息放到不同的路径中。路径的前缀“secret/”用来识别Vault后端，在这里使用了一个key-value的后端，名为“secret”。

私密信息的key和值是我们实际要写入Vault的内容。当Config Server要对外提供已写入的私密信息时，很重要的一点在于私密信息的key要和配置属性保持一致。

我们可以使用vault read命令校验私密信息是否已经写入Vault中：

```
$ vault read secret/application
```

```
Key                              Value
---                              -----
refresh_interval                 768h
spring.data.mongodb.password     s3cr3t
```

在将私密信息写入到指定路径的时候，需要注意每次往给定路径中写入时都会覆盖之前在该路径下写入的私密信息。例如，假设我们还想要往Vault的上述路径中写入MongoDB用户名，我们不能简单地写入spring.data.mongodb.username secret私密信息本身，如果这样做就会导致spring.data.mongodb.password私密信息丢失。我们需要同时将这两个属性写进去：

```
% vault write secret/application \
              spring.data.mongodb.password=s3cr3t \
              spring.data.mongodb.username=tacocloud
```

现在，我们已经往Vault中写入了一些私密信息。接下来，我们看一下如何让Vault作为Config Server的后端属性源。

在**Config Server**中启用**Vault**后端

为了将Vault添加为Config Server的后端，我们至少需要将Vault添加为激活的profile。在Config Server的application.yml文件中，将会如下所示：

```
spring:
  profiles:
    active:
    - vault
    - git
```

如上所示，vault和git profile均处于激活状态，允许Config Server同

时从Vault和Git获取配置。一般而言，我们会将敏感的配置属性写入Vault，对于不需要私密性的属性则继续使用Git作为后端。如果你希望将所有配置都写到Vault中或者没有必要使用Git后端，那么可以将spring.profiles.active设置为vault，完全放弃Git后端。

默认情况下，Config Server会假定Vault运行在localhost并监听8200端口。但是，我们可以在Config Server的配置中修改这种默认行为，如下所示：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          order: 2
        vault:
          host: vault.tacocloud.com
          port: 8200
          scheme: https
          order: 1
```

Config Server对Vault的默认假定都可以通过spring.cloud.config.server.vault.*相关的属性重写。在这里，我们告诉Config Server，Vault的API可以通过https://vault.tacocloud.com:8200来访问。

注意，我们保留了Git配置，假定Vault和Git分担了提供配置相关的职责。order属性表明Vault提供的私密属性要优先于Git提供的属性。

在配置完Config Server使用Vault作为后端之后，我们可以使用curl

命令伪装成一个客户端尝试一下：

```
[habuma:habuma]% curl localhost:8888/application/default | jq
{
  "timestamp": "2018-04-29T23:33:22.275+0000",
  "status": 400,
  "error": "Bad Request",
  "message": "Missing required header: X-Config-Token",
  "path": "/application/default"
}
```

噢，不！似乎出现问题了。实际上，这个错误表明Config Server提供来自 Vault的私密信息，但是请求中没有包含Vault token。

很重要的一点需要注意，对Vault的所有请求都要包含一个X-Vault-Token头信息。我们不会在Config Server本身中配置这个token，而是让每个Config Server客户端在向Config Server发送请求的时候在请求中包含X-Config-Token头信息。Config Server会接收到X-Config-Token头信息，然后将其转换成发送给Vault的X-Vault-Token头信息。

我们可以看到，因为在请求中缺少这个token，所以Config Server拒绝提供任何属性，甚至连Git中的属性都不可用了，因为在暴露私密的信息之前需要一个token。这是组合使用Vault和Git的一个有趣的副作用，除非提供一个合法的token，否则连Git属性都会被Config Server间接隐藏。

我们可以再尝试一下，在请求中添加一个X-Config-Token头信息：

```
$ curl localhost:8888/application/default
        -H"X-Config-Token: roottoken" | jq
```

请求中的这个X-Config-Token头信息应该会产生更好的结果，响应中将会包含我们写入到Vault中的私密信息。这里给出的token是在我们以开发模式启动Vault的时候设置的根token，但实际上Vault服务器创建的所有合法、未过期且具有访问Vault私密后端的token都是可以的。

在Config Server客户端设置Vault token

显然，在每个微服务中，我们不能使用curl来指定消费Config Server属性的token。相反，我们应该在服务应用的本地配置中添加一点配置信息：

```
spring:
  cloud:
    config:
      token: roottoken
```

spring.cloud.config.token属性会告诉Config Server客户端在每次向Config Server发送请求的时候都要带上给定的token。这个属性必须设置到应用的本地配置中（而不能存放到Config Server的Git或Vault中），Config Server才能够将其传递到Vault上，从而访问私密属性。

写入特定应用和特定profile的私密信息

在为Config Server提供服务的时候，写入application路径的属性适用于所有的应用，不管它们的名字是什么。如果我们想要写入针对给定应用的私密属性，就需要将路径中的application部分改成应用的名称。例如，如下的vault write命令会为名为ingredient-service的应用（通过其

spring.application.name属性指定）写入专有的私密信息：

```
$ vault write secret/ingredient-service \
              spring.data.mongodb.password=s3cr3t
```

类似的，如果我们不指定profile，写入Vault的私密信息就会成为默认profile属性的一部分。也就是说，不管哪个profile处于激活状态，客户端都能收到这些私密信息。我们可能想要将私密信息写入到特定的profile中，如下所示：

```
% vault write secret/application,production \
              spring.data.mongodb.password=s3cr3t \
              spring.data.mongodb.username=tacocloud
```

这种方式写入的私密信息只对激活profile为production的应用有效。

14.6 在运行时刷新配置属性

在编写本章的时候，我正在一架飞机上，因为维护问题，飞机被重新拉回了登机口。情况并不严重，你正在读本章的内容，就说明机械工程师的工作完成得还是很令人满意的。即便如此，关于飞机维护，最有意思的事情是它要求飞机必须要在地面上。如果飞机正在飞行，那么能做的事情就太少了。

相比之下，在《星球大战》（*Star Wars*）电影中，如果Luke Skywalker或Poe Dameron的X翼战机需要维护，舰载机械机器人（mech droid）就可以派上用场了，即使X翼战机正在作战，它也可以开展工作。

传统上，应用程序维护，包括配置更改，都需要重新部署或至少重新启动应用。可以说，由于缺少一个“机械机器人”来调整哪怕是最小的配置属性，因此我们每次都需要将应用程序拉回“登机口”。这对云原生应用来说是不可接受的。我们希望能够动态地更改配置属性，而不需要关闭应用程序。

幸运的是，Spring Cloud Config Server能够刷新正在运行的应用程序的配置属性，而不需要停机。一旦变更推送到支撑的Git仓库或Vault私密仓库，应用中的每个微服务就都可以立即通过以下两种方式的某一种进行刷新。

- 手动刷新：Config Server客户端启用一个特殊的“/actuator/refresh”端点，对每个服务的这个端点发送HTTP POST请求将会强制配置客户端从Config Server的后端检索最新的配置。
- 自动刷新：Git仓库上的提交hook会触发所有Config Server客户端服务的刷新操作。这涉及Spring Cloud的另一个项目，名为Spring Cloud Bus，它能够用于Config Server及其客户端之间的通信。

每种方案都有其优点和缺点。手动刷新能够更精确地控制服务何时更新最新配置，但是它需要向每个微服务实例发送一个HTTP请求。自动更新能够让应用中的每个微服务即时使用最新的配置，但它是由配置仓库的提交自动触发的，对于有些项目来说过于危险。

我们接下来看一下这两种方案，然后你就可以自行选择哪种方式更适合你的项目了。

14.6.1 手动刷新配置属性

在第16章中，我们将会介绍Spring Boot Actuator。它是Spring Boot的基本元素之一，能够探查应用运行时的状况并且允许对运行时进行一些有限的操作，比如修改日志级别。现在先看一个特殊的Actuator特性，只有配置为Spring Cloud Config Server客户端的应用，这个特性才有效。

当我们将应用设置为Config Server客户端的时候，自动配置功能会配置一个特殊的Actuator端点，用来刷新配置属性。为了使用该端点，在项目的构建文件中除了Config Client依赖，我们还需要添加Actuator starter依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

我们可以猜到，这项依赖也可以在Spring Initializr中通过选中Actuator复选框添加进来。

在Config Server客户端应用中添加Actuator之后，我们可以在任意时间发送HTTP POST请求到“/actuator/refresh”，通知它从后端仓库刷新配置属性。

我们看一下它是如何实现的。假设我们有一个带有@ConfigurationProperties注解的类，名为GreetingProps：

```
@ConfigurationProperties(prefix="greeting")
@Component
public class GreetingProps {
  private String message;

  public String getMessage() {
    return message;
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

另外，我们可以编写一个控制器类。GreetingProps会注入其中，当它在处理GET请求时，返回message属性的值：

```
@RestController
public class GreetingController {

  private final GreetingProps props;

  public GreetingController(GreetingProps props) {
    this.props = props;
  }

  @GetMapping("/hello")
  public String message() {
    return props.getMessage();
  }
}
```

在我们的Git配置仓库中有一个application.yml文件，含有如下的属性设置：

```
greeting:
  message: Hello World!
```

Config Server和这个简单的hello-world配置客户端运行起来之后，

我们对“/hello”发送HTTP GET请求，将会产生如下的响应：

```
$ curl localhost:8080/hello
Hello World!
```

现在，我们对Config Server和hello-world都不进行重启，而是修改application.yml文件并推送至后端Git仓库，这样greeting.message属性将会变成如下形式：

```
greeting:
  message: Hiya folks!
```

即便在Git中配置已经发生变化，如果我们发送GET请求到hello-world应用，得到的结果依然是“Hello World!”响应。但是，我们可以对刷新端点发送一个POST请求，强制使其刷新：

```
$ curl localhost:53419/actuator/refresh -X POST
["config.client.version","greeting.message"]
```

注意，响应中包含一个JSON数组，列出了发生变更的属性名。这个数组包含greeting.message属性，还包含config.client.version属性（当前配置对应的Git提交的哈希值）的变化。因为现在的配置基于一个新的Git提交，所以每当后端的配置仓库有变化时，这个值都会跟着变化。

POST请求的响应告诉我们greeting.message已经发生变化了。但是，真正的证据还是要靠再次向“/hello”路径发送GET请求：

```
$ curl localhost:8080/hello
Hiya folks!
```

无须重启应用，甚至无须重启Config Server，应用现在就能向我们提供greeting.message属性的全新值。

如果我们能够完全控制何时对配置属性进行更新，那么“/actuator/refresh”端点是很不错的选择。如果我们的应用由多个微服务组成（可能每个服务都有多个实例），那么将配置传播到所有服务可能是一项非常乏味的工作。接下来，我们看一下如何一次性地将配置变更自动用到所有服务上。

14.6.2 自动刷新配置属性

Config Server能够借助名为Spring Cloud Bus的Spring Cloud项目将配置变更自动通知到每个客户端，作为手动刷新应用中每个Config Server客户端属性的替代方案。图14.7阐述了它是如何运行的。

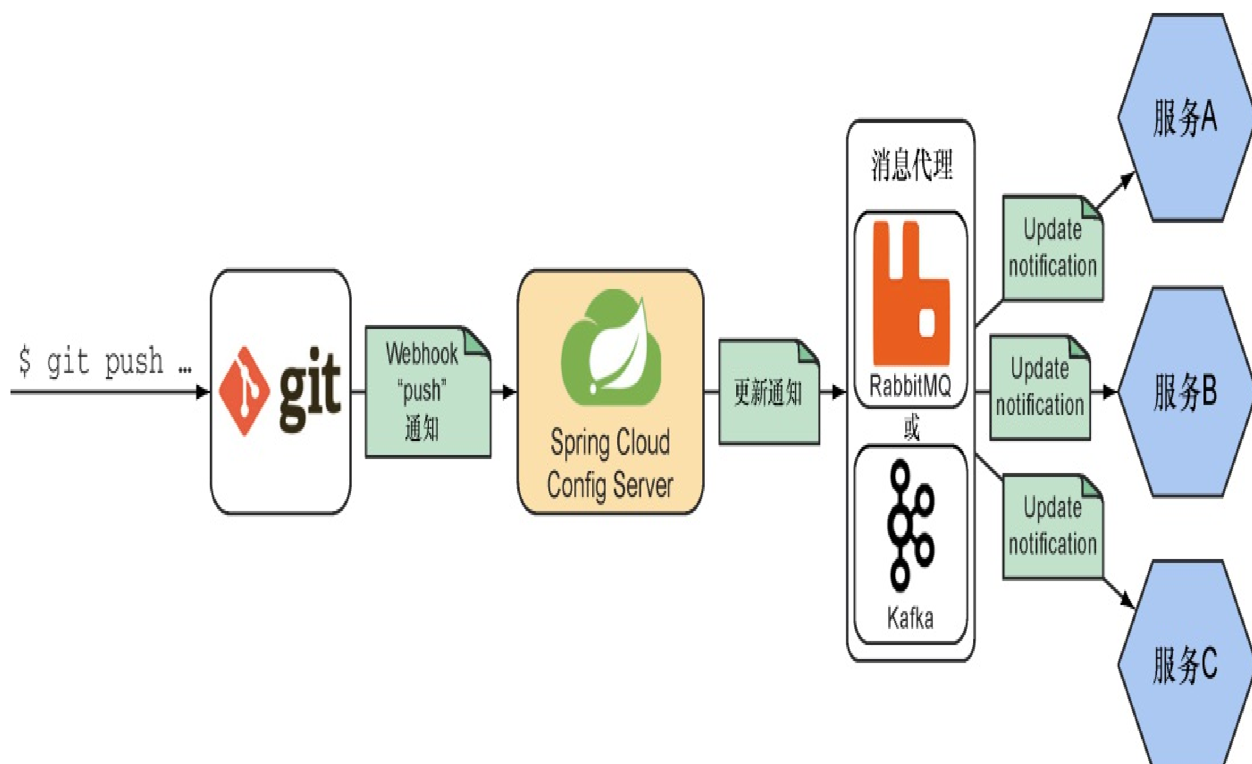


图14.7　Config Server与Spring Cloud Bus能够对应用广播变更（应用会在属性发生变化的时候，自动刷新它们的属性）

可以简要概括图14.7中的属性刷新流程。

- 在配置Git仓库上创建一个webhook，当Git仓库有任何变化（比如所有的推送）时，都会通知Config Server。很多的Git实现都支持webhook，比如GitHub、GitLab、Bitbucket和Gogs。
- Config Server会对webhook POST请求做出响应，借助某种消息代理以消息的方式广播该变更。
- 每个Config Server客户端应用订阅该通知，对通知消息做出响应，也就是会使用Config Server中的新属性值刷新它们的环境。

这样做的结果就是，在配置属性变更推送到后端的Git仓库之后，所有的Config Server客户端应用能够立即获取最新的配置属性值。

在使用Config Server的自动属性刷新功能时，会有多个部件在发挥作用。我们回顾一下要做的变更，这样对需要做的事情会有一个整体的了解。

- 我们需要有一个消息代理，用来处理Config Server及其客户端之间的消息传递，可以选择RabbitMQ或Kafka。
- 在后端Git仓库上需要创建一个webhook，将各种变更通知给Config Server。
- Config Server需要启用Config Server监控依赖（提供了处理Git仓库webhook请求的端点）以及RabbitMQ或Kafka的Spring Cloud Stream依赖（用于发布属性变更消息给代理）。
- 除非消息代理在本地按照默认设置运行，否则，我们要在Config Server及其所有的客户端上配置连接至代理的详细信息。
- 每个Config Server的客户端应用需要Spring Cloud Bus依赖。

假设预先需要的消息代理（不管是RabbitMQ、Kafka，还是你选择的其他方案）已经处于运行状态，并且为传送属性变更消息做好了准备，我们首先从将属性变更应用于Config Server开始，让它处理webhook的更新请求。

创建webhook

很多Git服务都支持创建webhook，从而能够将Git仓库的变更信息通知给应用，这些变更包括推送。不同实现之间创建webhook的操作有所差异，我们很难对它们一一描述。在这里，我会介绍如何为Gogs仓库创建webhook。

我选择Gogs的原因在于它非常易于在本地运行，并且支持将webhook POST用到本地运行的应用上（对于GitHub来说，这非常难以实现）。同时，在Gogs上创建webhook的过程与GitHub几乎完全相同，因此描述Gogs的过程能够间接让你知道为GitHub创建webhook都需要哪些步骤。

首先，在Web浏览器中访问配置仓库并点击Settings链接，如图14.8所示。（GitHub上Settings链接的位置略有差异，但是它们的外观很相似。）



图14.8　在Gogs或GitHub上点击Settings开始创建webhook

这会将我们带到仓库的设置页面，在左侧包含了一个设置分类的菜单。在菜单中选择Webhooks，将会出现如图14.9所示的页面。

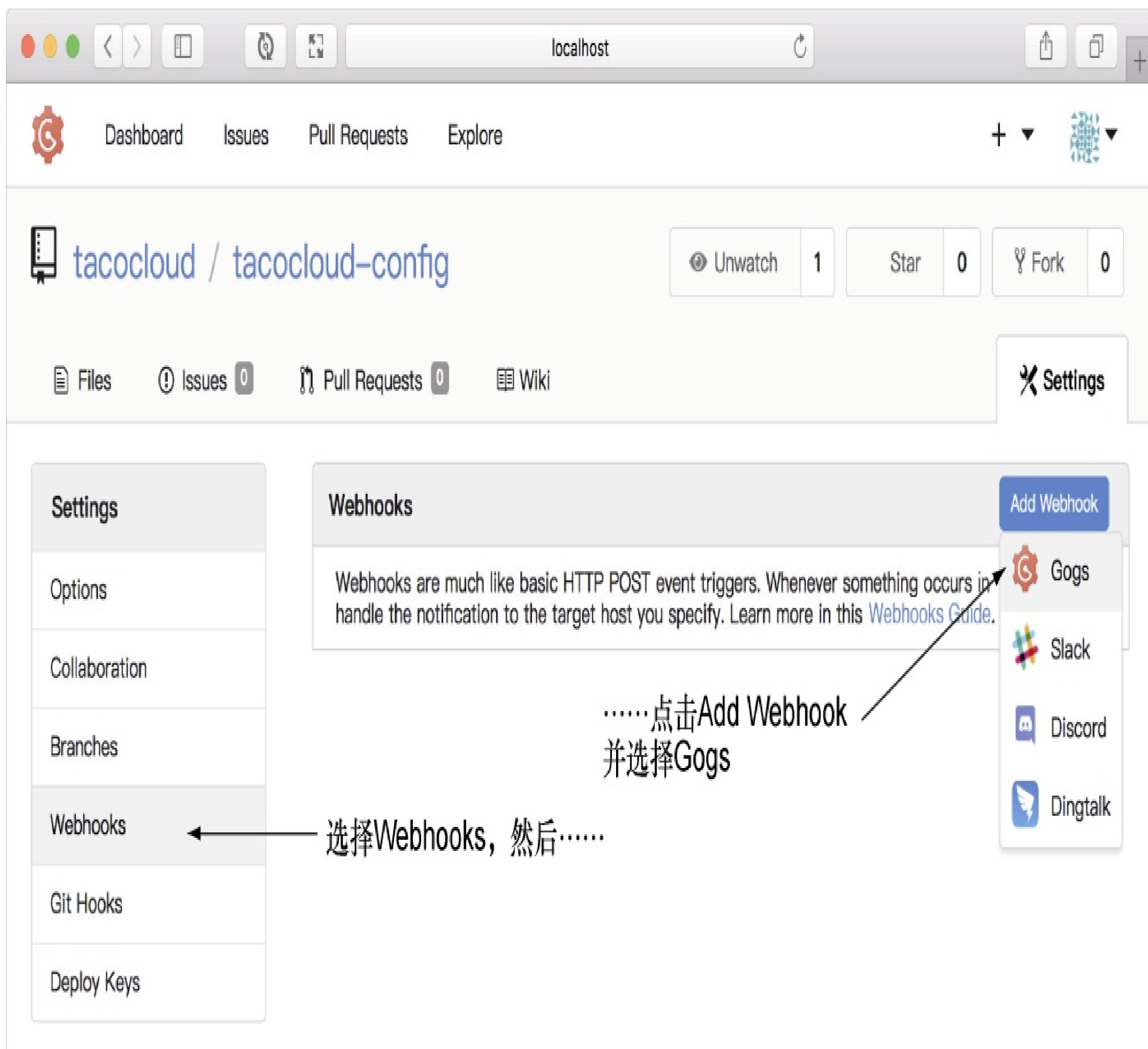


图14.9　Webhooks页面中的Add Webhook按钮会打开创建webhook的表单

在Webhooks设置页面，点击Add Webhook按钮，在Gogs中会生成一个下拉列表，用来选择不同类型的webhook。选择Gogs选项，如图14.9所示。这样，我们会看到一个创建新webhook的表单，如图14.10所示[1]。

Add Webhook表单有多个输入域，重要的是Payload URL和Content

Type。我们马上将会配置Config Server来处理webhook的POST请求。在实现该功能的时候，Config Server将会在“/monitor”路径下处理webhook请求。因此，我们需要将Payload URL输入域设置成引用Config Server的“/monitor”端点的URL。因为我是在一个Docker容器中运行Gogs的，所以在图14.10中将URL设置成http://host.docker.internal:8888/monitor，它的域名为host.docker.internal。这个域名让Gog服务器能够跨越容器的边界访问宿主机器上的Config Server[2]。

图14.10　创建webhook时需要指定Config Server的“/monitor”URL和JSON载荷

我还将Content Type输入域设置成了application/json。这一点非常重要，因为Config Server的“/monitor”端点并不支持Content Type的另一个选项application/x-www- form-urlencoded。

如果设置Secret输入域，就可以在webhook POST请求中新增一个名为X-Gogs-Signature（在GitHub中名为X-Hub-Signature）的头信息，包含给定私密信息的HMAC-SHA256摘要（在GitHub中是HMAC-SHA1）。此时，Config Server的“/monitor”端点并不识别这个签名头信息，因此我们可以将这个输入域设置为空。

最后，我们只关心配置仓库的推送请求，另外，我们当然希望这个webhook处于活跃状态，所以需要确保Just the push event单选框和Active复选框处于选中状态。点击表单底部的Add Webhook按钮，webhook就创建完成了。每当仓库有推送的时候，就会向Config Server发送POST请求。

现在，我们必须要启用Config Server的“/monitor”端点来处理这些请求。

在**Config Server**中处理**webhook**更新

要启用Config Server的“/monitor”端点非常简单，我们只需添加spring-cloud-config-monitor依赖到Config Server的配置文件即可。在Maven的pom.xml文件中，如下的依赖就会完成该项工作：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
```

```
  <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
```

这项依赖添加完成之后，自动配置功能会发挥作用，从而启用“/monitor”端点。但是，除非Config Server本身有广播变更通知的方法，否则不会带来任何好处。为了实现这一点，我们需要添加对Spring Cloud Stream的依赖。

Spring Cloud Stream是另一个Spring Cloud项目。借助它，我们能够创建通过底层绑定机制通信的服务，这种通信机制可能是RabbitMQ或Kafka。服务在编写的时候并不会关心如何使用这些通信机制，只是接受流中的数据，对其进行处理，并返回到流中，由下游的服务继续处理。

“/monitor”端点使用Spring Cloud Stream发布通知消息给参与的Config Server客户端。为了避免硬编码特定的消息实现，监控器会作为Spring Cloud Stream的源，发布消息到流中并让底层的绑定机制处理消息发送的特定功能。

如果使用RabbitMQ，就需要将Spring Cloud Stream RabbitMQ绑定依赖添加到Config Server的构建文件中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

如果你更喜欢Kafka，那么需要添加如下的Spring Cloud Stream Kafka依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

依赖准备就绪之后，Config Server几乎就可以参与属性自动刷新功能了。实际上，如果RabbitMQ或Kafka在本地运行并且使用默认配置，Config Server就已经可以运行了。如果消息代理在其他地方运行，而不是在localhost，或者使用了非默认端口，又或者我们修改了访问代理的凭证信息，就需要在Config Server本身的配置中添加一些属性了。

如果采用RabbitMQ绑定，那么application.yml中的如下条目可以用来重写默认值：

```
spring:
  rabbitmq:
    host: rabbit.tacocloud.com
    port: 5672
    username: tacocloud
    password: s3cr3t
```

虽然我们在这里设置了所有的属性，但是在你的RabbitMQ代理中只需要设置与默认值不同的属性即可。

如果使用Kafka，可以使用类似的属性：

```
spring:
  kafka:
    bootstrap-servers:
    - kafka.tacocloud.com:9092
    - kafka.tacocloud.com:9093
    - kafka.tacocloud.com:9094
```

你会发现，这些属性来源于第8章我们学习Kafka消息时的配置。实

际上，配置自动刷新功能的RabbitMQ和Kafka后端与在Spring中使用代理的其他场景非常相似。

创建**Gogs**的通知提取器

对于每个Git实现来说，webhook POST请求所携带的内容会有所不同。所以，对于“/monitor”端点来说，很重要的一点就是在处理webhook POST请求时能够理解不同的数据格式。在幕后，“/monitor”端点会有一组组件来检查POST请求，试图弄清楚请求来自哪种Git服务器，然后将请求数据映射为通用的通知类型，并发送至每个客户端。

Config Server对多个流行的Git实现提供了开箱即用的支持，比如GitHub、GitLab和Bitbucket。如果你使用其中的某一个实现，那么不需要任何额外的操作。在我编写本书的时候，Gogs还没有得到官方支持[3]。因此，使用Gogs作为Git实现的话，我们需要在项目中提供一个Gogs的通知提取器。

程序清单14.1为Taco Cloud集成Gogs时我所使用的通知提取器。

程序清单14.1　Gogs的通知提取器实现

```
package tacos.gogs;
import java.util.Collection;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import org.springframework.cloud.config.monitor.PropertyPathNotification;
import
     org.springframework.cloud.config.monitor.PropertyPathNotificationExtr
actor;
import org.springframework.core.Ordered;
```

```java
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;

@Component
@Order(Ordered.LOWEST_PRECEDENCE - 300)
public class GogsPropertyPathNotificationExtractor
    implements PropertyPathNotificationExtractor {

  @Override
  public PropertyPathNotification extract(
    MultiValueMap<String, String> headers,
    Map<String, Object> request) {
  if ("push".equals(headers.getFirst("X-Gogs-Event"))) {
    if (request.get("commits") instanceof Collection) {
      Set<String> paths = new HashSet<>();
      @SuppressWarnings("unchecked")
      Collection<Map<String, Object>> commits =
          (Collection<Map<String, Object>>) request
          .get("commits");
        for (Map<String, Object> commit : commits) {
          addAllPaths(paths, commit, "added");
          addAllPaths(paths, commit, "removed");
          addAllPaths(paths, commit, "modified");
        }
        if (!paths.isEmpty()) {
          return new PropertyPathNotification(
              paths.toArray(new String[0]));
        }
      }
    }
    return null;
  }

  private void addAllPaths(Set<String> paths,
                           Map<String, Object> commit,
                           String name) {
    @SuppressWarnings("unchecked")
    Collection<String> files =
            (Collection<String>) commit.get(name);
    if (files != null) {
      paths.addAll(files);
    }
  }
}
```

GogsPropertyPathNotificationExtractor如何运行的细节与我们的讨论没有太大关系，并且在Spring Cloud Config Server内置对Gogs的支持之后，就更加无关紧要了。所以，我不会对它进行过多的介绍，将它放在这里只是为了让你在使用Gogs的时候，可以作为参考。

在**Config Server**的客户端中启用自动刷新

在Config Server客户端启用属性的自动刷新比Config Server本身会更加简单。我们需要添加一项依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

这样会添加AMQP（如RabbitMQ）Spring Cloud Bus starter到构建文件中。

如果使用Kafka，就需要添加如下的依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>
```

对应的Spring Cloud Bus starter准备就绪之后，启动应用的时候，自动配置功能就会发挥作用，应用会自动将自己绑定到本地运行的RabbitMQ代理或Kafka集群上。如果你的RabbitMQ或Kafka在其他地方运行，那么我们需要在每个客户端应用上像Config Server本身那样配置它们的详细信息。

Config Server及其客户端都配置成了支持自动刷新。将它们启动起来，并对application.yml做一下修改（任意修改都可以），当将该文件提交至Git仓库的时候，我们会立即看到它在客户端应用中生效。

14.7 小结

- Spring Cloud Config Server提供了中心化的配置数据源，能够用于微服务架构应用中的所有微服务。
- Config Server提供的属性是通过后端的Git或Vault仓库维护的。
- 除了暴露给所有Config Server客户端的全局属性，Config Server还能提供特定profile和特定应用的配置。
- 敏感数据能够保持私密，这可以在后端Git仓库中通过对其进行加密来实现，也可以通过在Vault后端存储私密信息来实现。
- Config Server客户端能够借助手动或自动刷新得到新的属性，前者通过Actuator端点来实现，后者通过Spring Cloud Bus和Git webhooks来实现。

---

[1] GitHub没有可选webhook的下拉列表。在点击Add Webhook按钮之后，会直接出现创建webhook的表单。

[2] 在Docker容器中。localhost指的是容器本身，而不是Docker宿主机。

[3] 作者给Config Server项目提交了一个支持Gogs的pull request。在它合并进去之后，本书的这个章节就没有必要关注了。目前，作者的这个pull request经修改后，已经合并到了Config Server中。——译者注

第15章　处理失败和延迟

本章内容：

- 断路器模式简介
- 使用Hystrix处理失败和延迟
- 监控断路器
- 聚合断路器的指标

15.1　理解断路器模式

断路器模式是随着Michael Nygard的*Release It!*（第2版，Pragmatic Bookshelf，2018）一书流行起来的，解决了我们所编写的代码可能会失败的问题。很重要的一点在于，即便是失败，它也能够优雅地失败。这个强大的模式在微服务环境中会更加关键，因为在这种环境下避免跨调用堆栈产生级联失败非常重要。

相对来讲，断路器模式的理念很简单，非常类似于现实世界中的电

路断路器，这也是它得名的由来。在电路断路器中，当开关处于闭合位置时，电流能够流过断路器，为房间中的电灯、电视、电脑和其他设备供电。如果线路中出现故障，比如功率骤增，断路器就会打开，在电流损坏电子设备或房屋失火之前切断电流。

与之类似，软件中的断路器起初会处于关闭状态，允许进行方法的调用。如果因为某种原因，方法调用失败了（比如时间超出了定义的阈值），断路器就会打开，就不会对失败的方法再执行调用了。软件断路器的区别在于它提供了后备（fallback）行为和自校正功能。

如果被保护的方法在给定的失败阈值内发生了失败，那么可以调用一个后备方法代替它的位置。在断路器处于打开状态之后，几乎始终都会调用后备方法。处于打开状态的断路器偶尔会进入半开状态，并尝试调用发生失败的方法：如果依然失败，断路器就恢复为打开状态；如果调用成功，它会认为问题已经解决，断路器会回到闭合状态。图15.1阐述了软件断路器的流程。

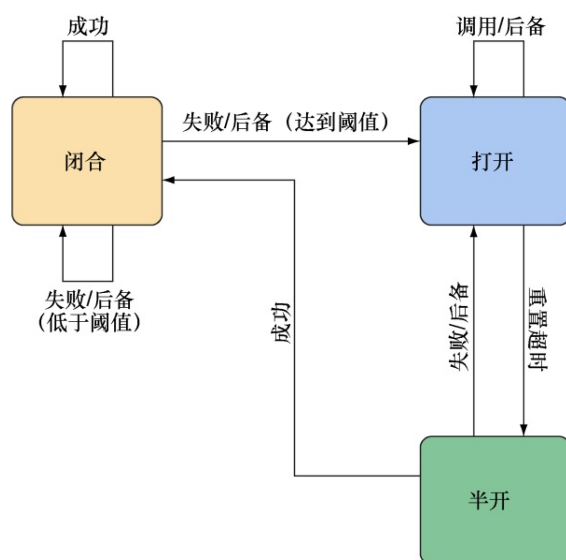


图15.1　断路器模式能够实现优雅的失败处理

我们可以将断路器想象成一个更加强大的try/catch。闭合的断路器类似于try代码块，而后备方法类似于catch代码块。与try/catch不同的地方在于，断路器非常智能，当预期方法频繁失败时它会绕过预期方法，始终调用后备方法。

按照我的阐述，断路器是应用到方法上的。这样，在给定的一个微服务中，很容易就能达到数十个（甚至更多）断路器。决定在代码的什么地方声明断路器其实就是识别哪些方法易于出现失败。如下的几类方法肯定是添加断路器的首选。

- 调用REST的方法：这些方法可能会因为远程服务不可用或者返回HTTP 500响应而失败。
- 执行数据库查询的方法：这些方法可能会因为数据库不响应或者模式变更破坏了应用而导致失败。
- 可能会比较慢的方法：它们不一定会失败，但是如果耗费太长时间才能完成工作就可能会被视为失败。

最后一项强调了除处理故障之外断路器的另一项收益。在微服务中，延迟也是非常重要的，某个执行缓慢的微服务不能拖慢整个微服务的性能，避免上游的服务产生级联延迟是非常重要的。

我们可以看到，断路器模式是在代码中优雅处理故障和延迟的强大方法。那么该如何将断路器用到我们的代码中呢？幸运的是，Netflix开源项目通过Hystrix为我们提供了答案。

Netflix Hystrix是断路器模式的Java实现。简而言之，Hystrix断路器实现为一个切面，会在目标方法发生失败的时候触发后备方法。为了实现断路器模式，这个切面还会跟踪目标方法失败的频率；如果失败率超过了某个阈值，那么所有的请求都会转发至后备方法。

关于**Hystrix**名称的一点逸事

当Netflix的开发人员为他们的断路器实现起名字的时候，他们想要这个名字能够体现出需要提供的弹性、防御能力和容错能力。最终，他们选择了Hystrix（Hystrix是古代豪猪的一种，豪猪是一种能够使用长刺进行自卫的动物）。此外，正如Hystrix FAQ中所解释的，这是一个听起来很酷的名称。当我们在15.3.1小节中查看Hystrix dashboard时，我们就会在项目的Logo位置处看到一个豪猪的图案。

Spring Cloud Netflix包含对Hystrix的支持，提供了一个简单的编程模型。Spring和Spring Boot开发人员都应该很熟悉这个模型。为方法添加@HystrixCommand注解并提供一个后备方法，就可以为该方法声明断路器。下面让我们看看如何在Taco Cloud代码中声明断路器，从而优雅地使用Hystrix来处理失败。

15.2 声明断路器

在声明断路器之前，我们需要添加Spring Cloud Netflix Hystrix starter依赖到每个服务的构建文件中。在Maven pom.xml文件中，依赖如下所示：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

作为Spring Cloud套件的一部分，我们需要在构建文件中声明Spring Cloud release train的依赖管理。在我编写本书的时候，最新的release train版本为Finchley.SR1。所以，应该将Spring Cloud的版本设置为一个属性，如下的条目应该出现在pom.xml文件的<dependencyManagement>代码块中：

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

> 注意：在创建项目的时候，starter依赖也可以在Initializr中通过名为Hystrix的复选框来进行添加。如果使用Initializr添加Hystrix到项目的构建文件中，那么依赖管理代码块会自动创建。

Hystrix starter就绪之后，接下来的事情就是启用Hystrix。为了实现这一点，我们可以在应用的主配置类上添加@EnableHystrix。例如，为了在配料服务上启用Hystrix，我们可以按照如下的方式为IngredientServiceApplication添加注解：

```
@SpringBootApplication
@EnableHystrix
public class IngredientServiceApplication {
    ...
}
```

这样，在我们的应用中就启用Hystrix了，也就意味着声明断路器的所有准备工作都做完了。在我们的代码中还没有声明任何一个断路器，这时@HystrixCommand注解就能够发挥作用了。

任何使用@HystrixCommand注解的方法都会为其声明一个断路器切面。例如，如下的方法使用支持负载均衡的RestTemplate从配料服务中获取一个Ingredient对象的列表：

```
public Iterable<Ingredient> getAllIngredients() {
  ParameterizedTypeReference<List<Ingredient>> stringList =
      new ParameterizedTypeReference<List<Ingredient>>() {};
  return rest.exchange(
      "http://ingredient-service/ingredients", HttpMethod.GET,
      HttpEntity.EMPTY, stringList).getBody();
}
```

对exchange()的调用可能会遇到问题。如果Eureka没有注册名为ingredient-service的服务或者由于某种原因请求失败了，那么将会抛出RestClientException（非检查型异常）。因为异常没有在try/catch代码块中进行处理，所以调用者必须要处理这个异常。如果调用者不处理，那

么它将沿着调用栈往上抛出；如果它根本没有得到处理，那么这个错误会级联到所有上游微服务或客户端。

在任何应用中，未捕获的异常都是一项艰巨的挑战，在微服务中尤为如此。当遇到失败的时候，微服务应该应用维加斯规则（Vegas Rule）：在微服务中发生的事情，就留在微服务中。在getAllIngredients()方法上声明断路器将会满足该规则。

按照最少的要求，我们只需要为该方法添加@HystrixCommand注解并为其提供一个后备方法即可。首先，我们添加@HystrixCommand注解到getAllIngredients()方法上：

```
@HystrixCommand(fallbackMethod="getDefaultIngredients")
public Iterable<Ingredient> getAllIngredients() {
  ...
}
```

断路器为getAllIngredients()提供了失败防护，所以在遇到失败时它是安全的。如果由于某种原因getAllIngredients()抛出了未捕获的异常，那么断路器将会捕获它们并将方法调用重定向到名为getDefaultIngredients()的方法上。

你可以让后备方法做任何事情，但是它们的本意是当原始的方法无法履行职责时提供后备行为。后备行为方法的唯一规则是它们要与原始方法具有相同的签名（除了方法名称之外）。

为了满足该要求，getAllIngredients()也不能接受任何参数并要返回List<Ingredient>。如下的getAllIngredients()实现满足该规则，并且返回

一个默认的配料列表：

```
private Iterable<Ingredient> getDefaultIngredients() {
  List<Ingredient> ingredients = new ArrayList<>();
  ingredients.add(new Ingredient(
        "FLTO", "Flour Tortilla", Ingredient.Type.WRAP));
  ingredients.add(new Ingredient(
        "GRBF", "Ground Beef", Ingredient.Type.PROTEIN));
  ingredients.add(new Ingredient(
        "CHED", "Shredded Cheddar", Ingredient.Type.CHEESE));
  return ingredients;
}
```

现在，如果因为某种原因导致getAllIngredients()失败，那么断路器将会调用备用的getDefaultIngredients()，调用者将会接收到默认的配料列表（尽管非常有限）。

你可能会想，如果备用方法本身有断路器又会怎样呢。尽管按照我们的写法，getDefaultIngredients()几乎不可能会出问题，但是更有意思的是getDefaultIngredients()可能会有潜在的失败点。如果是这样，那么我们可以在getDefaultIngredients()上添加@HystrixCommand注解并提供另一个备用方法。实际上，需要的话，我们可以堆积任意数量的备用方法。唯一的要求就是必须要在后备方法的底部有一个不会失败的方法，该方法不需要使用断路器。

15.2.1 缓解延迟

断路器还能缓解延迟。如果某个方法需要较长的时间才能返回，断路器会将它设置为超时。默认情况下，所有带有@HystrixCommand注解的方法都会在1秒之后超时，并调用它们所声明的后备方法。这意味

着，如果因为某种原因配料服务响应缓慢，那么对getAllIngredients()调用会在1秒之后超时，而且会调用getDefaultIngredients()作为替代方案。

1秒超时是一个合理的默认值，适用于大多数的场景。我们也可以通过Hystrix命令属性将其调整为更大或更小的限制值。设置Hystrix命令属性可以通过@HystrixCommand注解的 commandProperties属性来实现。commandProperties属性是一个或多个@HystrixProperty注解所组成的数组，指定了要设置的属性名和值[1]。

为了调整断路器的超时值，我们需要设置Hystrix命令属性execution.isolation.thread.timeoutInMilliseconds。例如，为了将getAllIngredients()的超时时间更加严格地设置为0.5秒，那么我们可以将超时设置为500，如下所示：

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="execution.isolation.thread.timeoutInMilliseconds",
            value="500")
    })
public Iterable<Ingredient> getAllIngredients() {
  ...
}
```

这里设置的值是毫秒数。如果我们希望放松限制，那么可以将其设置成一个更大的值。或者，如果你认为这里不应该使用超时功能，那么可以将execution.timeout.enabled属性设置为false，直接将超时功能移除：

```
@HystrixCommand(
```

```
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="execution.timeout.enabled",
            value="false")
    })
public Iterable<Ingredient> getAllIngredients() {
  ...
}
```

将execution.timeout.enabled为false的话就没有延迟防护了。在本例中，getAllIngredients()方法不管是耗用1秒、10秒还是30分钟，它都不会超时。这可能会导致级联的延迟效果，所以在禁用执行超时的时候要非常小心。

15.2.2 管理断路器的阈值

默认情况下，如果断路器保护的方法调用超过20次，而且50%以上的调用在10秒的时间内发生失败，那么断路器就会进入打开状态。所有后续的调用都将会由后备方法处理。在5秒之后，断路器进入半开状态，将会再次尝试调用原始的方法。

我们可以通过设置Hystrix命令属性调整失败和重试的阈值。如下的命令属性将会影响断路器的行为。

- circuitBreaker.requestVolumeThreshold：在给定的时间范围内，方法应该被调用的次数。
- circuitBreaker.errorThresholdPercentage：在给定的时间范围内，方法调用产生失败的百分比。
- metrics.rollingStats.timeInMilliseconds：控制请求量和错误百分比的

滚动时间周期。

- circuitBreaker.sleepWindowInMilliseconds：处于打开状态的断路器要经过多长时间才会进入半开状态，进入半开状态之后，将会再次尝试失败的原始方法。

如果在metrics.rollingState.timeInMilliseconds设定的时间范围内超出了circuitBreaker.requestVolumeThreshold和circuitBreaker.errorThresholdPercentage设置的值，那么断路器将会进入打开状态。在circuitBreaker.sleepWindowInMilliseconds限定的时间范围内，它会一直处于打开状态，在此之后将进入半开状态，进入半开状态之后，将会再次尝试失败的原始方法。

例如，我们调整失败的设置：将其变更为在20秒的时间范围内调用超过30次且失败率超过25%。为了实现这一点，我们需要按照如下的方式调整Hystrix命令属性：

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="circuitBreaker.requestVolumeThreshold",
            value="30"),
        @HystrixProperty(
            name="circuitBreaker.errorThresholdPercentage",
            value="25"),
        @HystrixProperty(
            name="metrics.rollingStats.timeInMilliseconds",
            value="20000")
    })
public List<Ingredient> getAllIngredients() {
  // ...
}
```

另外，我们还决定处于打开状态之后断路器必须保持1分钟，然后

才进入半开状态，那么我们还需要设置circuitBreaker.sleepWindowInMilliseconds命令属性：

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        ...
        @HystrixProperty(
            name="circuitBreaker.sleepWindowInMilliseconds",
            value="60000")
    })
```

除了优雅地处理方法调用失败和延迟之外，Hystrix还为应用中的每个断路器提供了一个指标流。接下来，我们看一下如何通过Hystrix流监控启用Hystrix功能的应用的监控状况。

15.3　监控失败

每当断路器保护的方法被调用时，它都会收集一些调用相关的数据，并将其发布到一个HTTP流中，这些数据可以实时监控正在运行中的应用的健康状况。在每个断路器收集的数据中，Hystrix流包括如下内容：

- 方法被调用了多少次；
- 调用成功了多少次；
- 后备方法调用了多少次；
- 方法超时了多少次。

Hystrix流是由Actuator端点提供的。在第16章中，我们会更详细地讨论Actuator，现在只需要将Actuator依赖添加到所有服务的构建文件

中，以便于启用Hystrix流即可。在Maven pom.xml文件中，如下的starter依赖会将Actuator添加到项目中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Hystrix流端点会通过“/actuator/hystrix.stream”路径对外暴露。默认情况下，大多数的端点都是禁用的。我们可以通过在每个应用的application.yml文件中添加如下的配置启用Hystrix端点：

```
management:
  endpoints:
    web:
      exposure:
        include: hystrix.stream
```

我们还可以将management:endpoints:web:exposure:include属性放到Config Server对外提供配置属性的application.yml文件中，这样全局所有的服务就都可以使用它了。

应用启动之后，将会暴露Hystrix流（这个流可以被任意的REST端点消费）。在编写自定义的REST端点之前，我们需要注意HTTP流的每个条目都包含了各种类型的JSON数据，客户端需要大量的工作才能解析这些数据。尽管编写自定义的Hystrix流展现层并非不可能完成的任务，但是在花费大量工夫编写自己的dashboard之前我们可以考虑一下使用Hystrix的dashboard。

15.3.1　Hystrix Dashboard简介

要使用Hystrix Dashboard，我们首先创建一个Spring Boot应用并添加对Hystrix dashboard starter的依赖。如果使用Spring Boot Initializr来创建项目，就可以选择Hystrix Dashboard复选框；否则，我们需要添加如下的<dependency>到项目的Maven pom.xml文件中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

项目初始化之后，我们可以通过为主配置类添加@EnableHystrixDashboard注解来启用Hystrix dashboard：

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
  public static void main(String[] args) {
    SpringApplication.run(HystrixDashboardApplication.class, args);
  }
}
```

在开发阶段，我们可能会让Hystrix Dashboard与其他服务一起在本地机器运行，还包括Eureka和Config Server。因此，为了避免端口冲突，我们需要为Hystrix Dashboard选取一个唯一的端口。在Dashboard应用的application.yml文件中，我们可以将server.port设置成任意唯一的值，我通常会将其设置为7979，如下所示：

```
server:
  port: 7979
```

现在，我们就可以启动Hystrix Dashboard并查看其效果了。运行之

后，打开浏览器并访问http://localhost:7979/hystrix，我们将会看到如图15.2所示的Hystrix Dashboard主页。

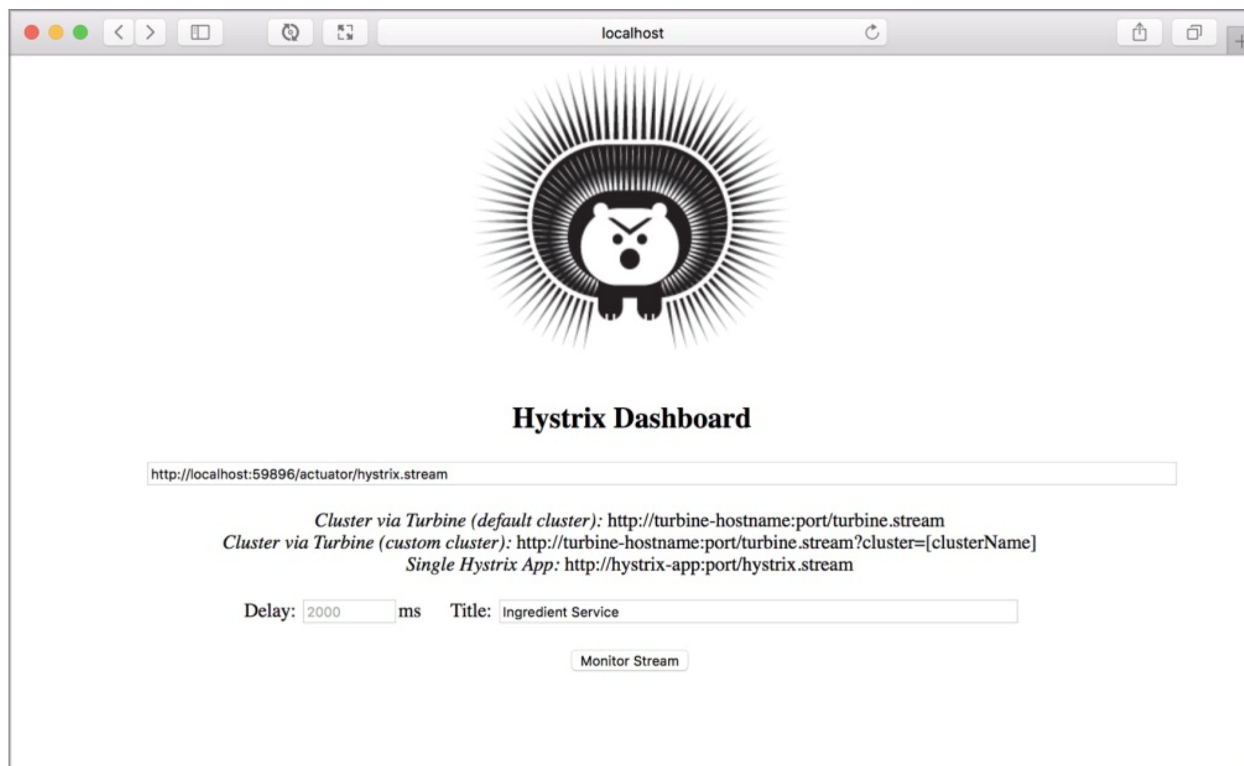


图15.2　Hystrix Dashboard主页

关于Hystrix Dashboard主页，你首先会注意到的可能是logo（Hystrix项目的卡通豪猪吉祥物）。要查看Hystrix流，我们需要输入某个服务应用的Hystrix流URL到文本域中。比如，配料服务在localhost运行并且监听59896（这是因为将server.port设置成了0），那么我们可以在文本框中输入“http://localhost:59896/actuator/hystrix.stream”。

在Hystrix流监视器中，我们可以设置延迟和标题。延迟的默认值是2秒，指的是轮询周期的间隔，它实际上会延缓流。标题输入域的值会以标题的形式显示在监控页中。对于我们的需求来说，默认值就可以

了。

点击Monitor Stream按钮，我们就可以进入Hystrix流的监视器页面了，如图15.3所示。

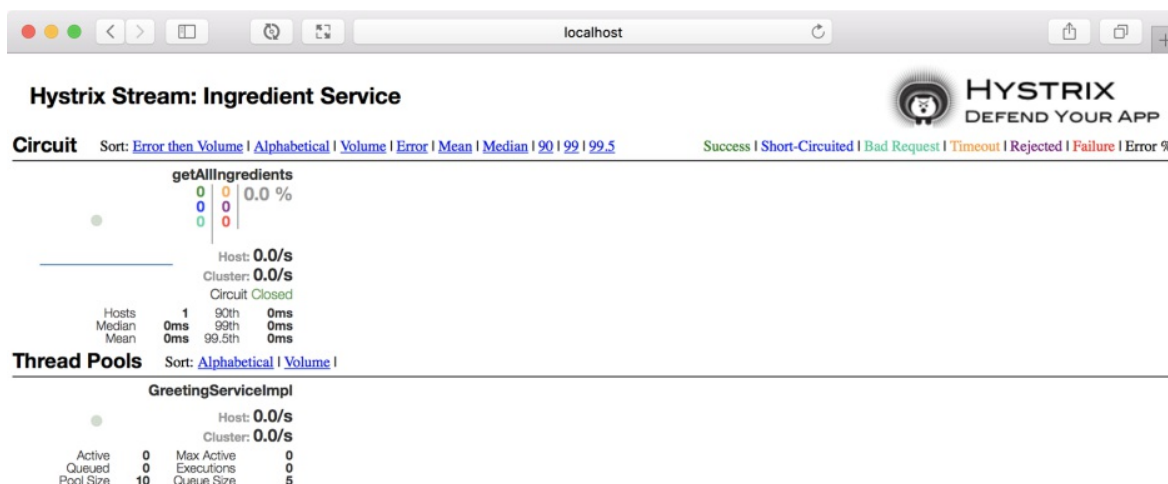


图15.3　Hystrix流监控页面会显示每个应用的断路器指标

每个断路器都会显示为一个图表并且还带有一些有用的指标数据。图15.3中只显示了getAllIngredients()的断路器，因为到目前为止我们只定义了这一个断路器。

如果看不到断路器的任何图表，只是看到一个单词“Loading”，那么可能是断路器方法都还没有被调用。我们必须向服务发送一个请求，触发断路器保护该方法，这样方法的断路器指标才会显示在Dashboard上。我近距离观察了一个断路器的监视器（见图15.4），并对其中显示的所有数据都进行了标注。

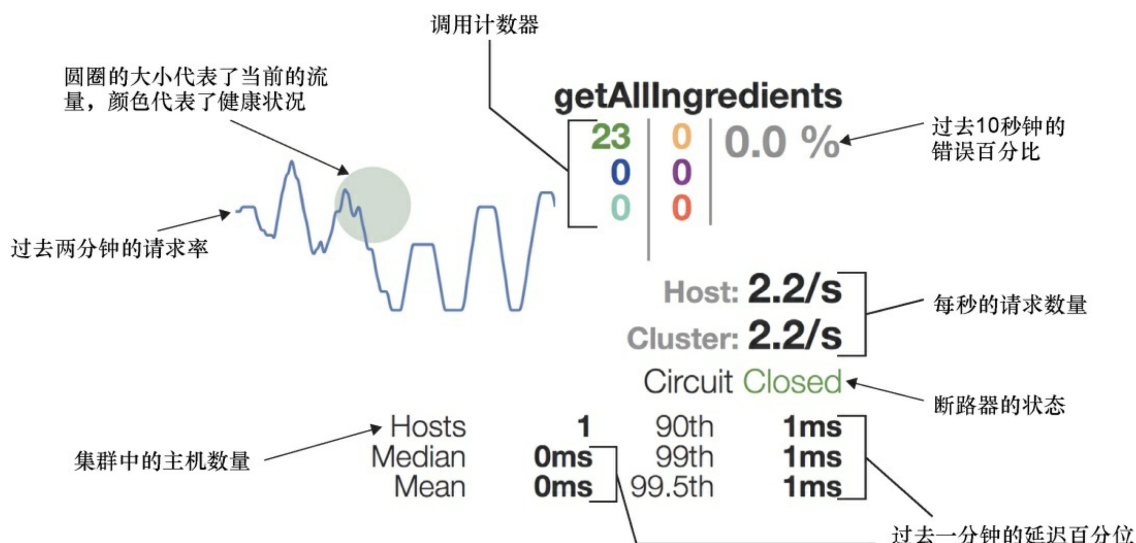


图15.4　每个断路器的监视器都提供了该断路器当前状态的有用信息

在监视器中，最引人注目的是左上角的图表。折线图代表了指定方法过去两分钟的流量，简要显示了该方法的繁忙情况。

折线图的背景是一个大小和颜色会出现波动的圆圈。圆圈的大小表示当前的流量，圆圈越大，流量越大。圆圈的颜色表示它的健康状况：绿色表示健康的断路器，黄色表示偶尔发生故障的断路器，红色表示故障断路器。

在监视器的右上角，以3列的形式显示各种计数器。在最左边的一列中，从上到下，第一个数字（绿色——在本书的电子版中会看出各种颜色）表示当前成功调用的数量，第二个数字（蓝色）表示短路请求的数量，最后一个数字（蓝绿色）表示错误请求的数量。中间一列显示超时请求的数量（黄色）、线程池拒绝的数量（紫色）和失败请求的数量（红色）。第三列显示过去10秒内错误的百分率。

计数器下面有两个数字，代表每秒主机和集群的请求数量。这两个

请求率下面是断路器的状态。监视器的底部显示了延迟的中位数和平均值，以及第90、99和99.5百分位的延迟。

15.3.2　理解Hystrix的线程模型

假设某个方法要耗费大量的时间才能完成其任务。这个方法可能向其他的服务发起了HTTP请求，而该服务响应很慢。在服务响应之前，Hystrix会阻塞线程，等待响应。

如果这个方法执行时与调用者在同一个线程上下文中，那么调用者将会一直在这个长时间运行的方法上进行等待。另外，如果被阻塞的线程来自一组数量有限的线程集，比如Tomcat的请求处理线程，而且这种情况一直持续，那么当所有线程耗尽并全部等待响应时，就会影响到可扩展性。

为了避免这种现象，Hystrix会为每项依赖（比如，带有一个或多个Hystrix命令方法的每个Spring bean）指派一个线程池。当Hystrix命令调用时，它将会在来自Hystrix托管的线程池的某个线程中执行，这样会将其与调用者线程隔离开。如果被调用的方法要执行较长时间，就能够允许调用线程不用一直等待，将潜在的线程耗尽隔离在Hystrix托管的线程池中。

你可能会发现在图15.3中除了断路器的监视器外，在页面底部还有另一个监视器，位于“Thread Pools”标题之下。这个区域是Hystrix托管的每个线程池的监视器。图15.5展示了一个线程池监视器，并对其中的数

据进行了标注。

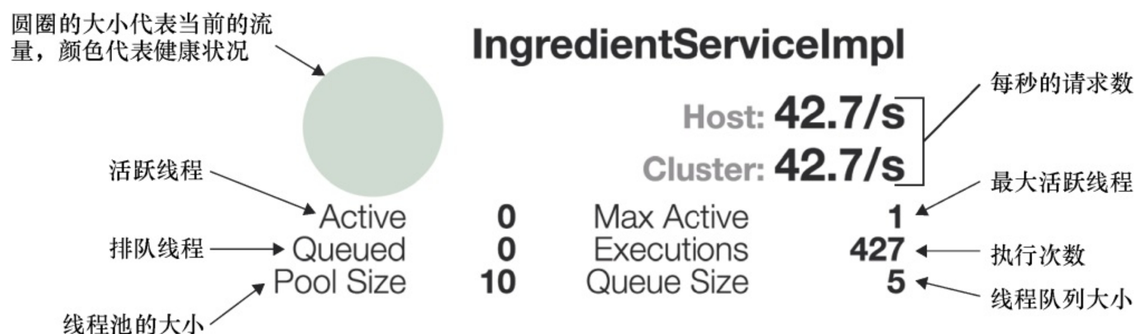


图15.5　线程池监视器显示Hystrix托管的线程池的重要统计信息

与断路器的监视器类似，每个线程池监视器在左上角都含有一个圆圈。圆圈的大小和颜色代表了线程池的活跃状态以及它的健康状况。与断路器的监视器不同的是，线程池的监视器没有显示过去几分钟线程池活动的折线图。

右上角显示线程池的名称，其下方是线程池中的线程每秒钟处理请求的数量。线程池监视器的左下角显示如下信息。

- 活跃线程：当前活跃线程的数量。
- 排队线程：当前有多少线程在排队。默认情况下，队列功能是禁用的，所以这个值始终为0。
- 线程池的大小：线程池中有多少线程。

在右下角显示线程池的其他信息：

- 最大活跃线程：在当前的采样周期中，活跃线程的最大数量。
- 执行次数：线程池中的线程被调用执行Hystrix命令的次数。
- 线程队列大小：线程池队列的大小。线程队列功能默认是禁用的，所以这个值没有什么意义。

值得一提的是，作为Hystrix线程池的替代方案，我们可以选择使用信号量隔离（semaphore isolation）。然而，信号量隔离是Hystrix的更高级用法，超出了本章的范围。有关它的更多信息，请参考Hystrix的文档。

现在，我们已经看到了Hystrix dashboard是如何运行的。接下来，我们考虑一下如何处理多个断路器数据流，以及如何将它们聚合到一个流中，以便在Hystrix dashboard中查看。

15.4 聚合多个Hystrix流

Hystrix dashboard一次只能监控一个流。因为每个微服务实例都发布它们自己的Hystrix，所以几乎不可能对整个应用的健康状况历史有一个整体的了解。

幸运的是，Netflix的另一个项目Turbine提供了将所有微服务的所有Hystrix流聚合到一个Hystrix流中的办法，这样Hystrix dashboard就能对其进行监控了。Spring Cloud Netflix支持以类似于创建其他Spring Cloud服务的方式创建Turbine服务。要创建Turbine服务，我们需要创建一个新的Spring Boot项目并将Turbine starter依赖添加到构建文件中：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-turbine</artifactId>
</dependency>
```

注意：作为一个新项目，最简单的方式是在创建新Spring Boot项目的时候在Initializr中选中Turbine复选框。

在创建完新项目之后，我们需要启用Turbine。为了实现这一点，我们需要在应用的主配置类上添加@EnableTurbine注解：

```
@SpringBootApplication
@EnableTurbine
public class TurbineServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(TurbineServerApplication.class, args);
  }
}
```

在开发阶段，我们会和Taco Cloud应用的其他服务一起在本地运行Turbine。为了避免端口冲突，我们需要为Turbine选择一个唯一的端口，这样就不会与其他的服务产生冲突了。你可以选择任意的端口，不过我倾向于选择8989：

```
server:
  port: 8989
```

Turbine会消费多个微服务的流并将它们的断路器指标合并到一个流中。它会作为Eureka的客户端，发现那些需要聚合到自己的流的服务。但是，Turbine并不想聚合Eureka中注册的所有流，所以我们必须配置Turbine，告诉它都要使用哪些服务。

turbine.app-config属性会接受一个由逗号分隔的服务名称列表，

Turbine会在Eureka中查找它们并聚合它们的Hystrix流。对Taco Cloud应用来讲，我们需要注册在Eureka中的4个服务，即ingredient-service、taco-service、order-service和user-service。如下的application.yml配置条目展现了如何设置turbine.app-config：

```
turbine:
  app-config: ingredient-service,taco-service,order-service,user-service
  cluster-name-expression: "'default'"
```

注意，除了turbine.app-config之外，我们还将turbine.cluster-name-expression属性设置成了“'default'”。这表明Turbine会收集名为default的集群中的所有聚合流。设置这个属性是非常重要的，否则Turbine中不会包含任何特定应用的聚合流数据。

现在，启动Turbine服务器并让Hystrix dashboard访问http://localhost:8989/ turbine.stream地址上的流，特定应用的所有断路器都将会展现在断路器dashboard上，如图15.6所示。

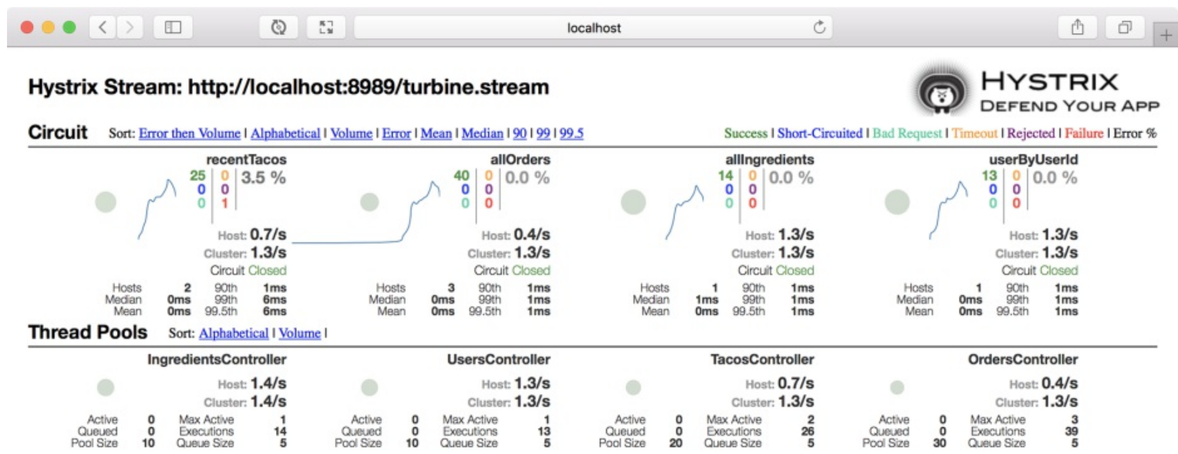


图15.6　当访问聚合的Turbine流时Hystrix dashboard会显示所有服务的所有断路器

Hystrix dashboard能够展现所有服务的所有断路器要归功于Turbine。这样，我们就能够一站式地监控Taco Cloud应用所有断路器的健康状况了。

15.5　小结

- 断路器模式能够优雅地进行失败处理。
- Hystrix实现了断路器模式，能够在某个方法失败或执行太慢的时候启用后备行为。
- Hystrix提供的每个断路器都会以数据流的方式发布指标信息，以便于监控应用的健康状况。
- Hystrix可以被Hystrix Dashboard消费，这是一个可视化断路器指标的Web应用。
- Turbine能够将多个应用的Hystrix流聚合到一个流中，以便在Hystrix Dashboard中统一进行可视化展现。

---

[1]　不知道你是不是像我一样，觉得以注解的形式为其他注解设置属性的方法有点诡异。不管是不是觉得诡异，这就是目前的做法。

第5部分 部署Spring

在第 5 部分中，我们将会介绍如何为应用的部署做好准备，并且会学习如何进行部署。第16章介绍Spring Boot Actuator。这是Spring Boot的一个扩展，以REST端点和JMX MBean的形式暴露正在运行中的应用的内部状况。在第17章中，我们将会看到如何使用Spring Boot Admin。它基于Actuator提供了一个用户友好的、基于浏览器的管理型应用。我们将会看到如何注册客户端应用以及如何保护Admin Server。第18章将讨论如何以JMX MBean的形式暴露和消费Spring bean。在最后的第19章中，我们将会看到如何将Spring应用部署到各种生产环境中。部署过基于Java应用的人可能认为这轻而易举，但是Spring Boot和相关的Spring项目有很多特性，使得Spring Boot应用的部署有些不同。

第16章　使用Spring Boot Actuator

本章内容：

- 在Spring Boot项目中启用Actuator
- 探索Actuator的端点
- 自定义Actuator
- 保护Actuator

你有没有试图猜测包装好的礼物盒中到底有什么东西的经历？你可能摇晃、掂量或者用尺子测量它。对于里面有什么东西，你可能会有一个确定的想法。但是，在真正将它打开之前，我们无法完全确定。

运行中的应用有点像包装好的礼物。你可以探测一下它，然后对里面的运行状况做出一个合理的猜测。但是，我们该如何确定呢？如果能有一种方式让我们窥探运行中的应用，假设我们能够查看它的行为、检查它的健康状况，甚至触发影响它运行的各种操作，那就太好了！

在本章中，我们将会讨论Spring Boot的Actuator。Actuator提供了生产环境可用的特性，包括监控Spring Boot应用和获取它的各种指标。Actuator的特性是通过各种端点提供的，这些端点可以通过HTTP调用，也可以通过JMX MBean来使用。在本章中，我们主要关注HTTP端点，而对JMX端点的介绍留到第19章。

16.1　Actuator概览

在机器领域中，执行机构（Actuator）指的是负责控制和移动装置的组件。在Spring Boot应用中，Spring Boot Actuator扮演了相同的角色，它能够让我们看到一个运行中的应用的内部状况，而且能够在一定程度上控制应用的行为。

通过Actuator暴露的端点，我们可以获取一个正在运行中的应用的内部状态。

- 在应用环境中，都有哪些可用的配置属性？
- 在应用中，各个源码包的日志级别是什么？
- 应用消耗了多少内存？
- 给定的HTTP端点被请求了多少次？
- 应用本身以及与它协作的外部服务的健康状况如何？

为了在Spring Boot应用中启用Actuator，我们需要在构建文件中添加对Actuator starter的依赖。在Spring Boot应用的Maven pom.xml文件中，添加如下的<dependency>条目就能完成该任务：

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

将Actuator starter添加到项目的构建文件中之后，应用就会具备一些开箱即用的Actuator端点，其中一部分如表16.1所示。

表**16.1**　探查运行中**Spring Boot**应用的状态并对其进行操作的**Actuator**端点

HTTP方法	路径	描述	默认是否启用
GET	/auditevents	生成所有已触发的审计事件的报告	否
GET	/beans	描述Spring应用上下文中所有的bean	否
GET	/conditions	生成一个自动配置条件通过或失败的报告，这些条件会指导应用上下文中bean的创建	否
GET	/configprops	描述所有的配置属性以及当前的值	否
GET、POST、DELETE	/env	生成Spring应用可用的所有属性源以及可用属性的报告	否
GET	/env/{toMatch}	描述某个环境属性的值	否
GET	/health	返回聚合的应用健康状态，可能的话，还会包含外部依赖应用的健康状态	是

GET	/heapdump	下载堆dump文件	否
GET	/httptrace	生成最近100个请求的跟踪结果	否
GET	/info	返回开发人员定义的关于该应用的信息	是
GET	/loggers	生成应用中源码包的列表，其中会包含配置的以及生效的日志级别	否
GET、POST	/loggers/{name}	返回指定logger配置的和生效的日志级别，生效的日志级别可以通过POST请求修改	否
GET	/mappings	生成所有HTTP映射及其对应处理器方法的报告	否
GET	/metrics	返回所有指标分类的列表	否
GET	/metrics/{name}	返回给定指标的多维度值集	否
GET	/scheduledtasks	列出所有的调度任务	否
GET	/threaddump	返回所有应用线程的报告	否

除了基于HTTP的端点之外，表16.1中除“/heapdump”的其他端点都

以JMX MBean的形式对外暴露了出来。我们将会在第19章学习JMX侧的Actuator。

16.1.1 配置Actuator的基础路径

默认情况下，表16.1中所有端点的路径都会带有“/actuator”。这意味着，如果我们想要通过Actuator获取应用的健康信息，那么向“/actuator/health”发送GET请求就能返回所需的信息。

Actuator的前缀可以通过设置management.endpoint.web.base-path属性来修改。例如，想要将前缀设置为“/management”，那么我们可以通过如下的方式来设置management. endpoint.web.base-path属性：

```
management:
  endpoints:
    web:
      base-path: /management
```

按照上述属性，要获取应用的健康信息，我们需要向“/management/health”发送GET请求。

16.1.2 启用和禁用Actuator端点

你可能已经发现，默认情况下，只有“/health”和“/info”端点是启用的。大多数Actuator端点会携带敏感信息，所以应该保护起来。我们可以使用Spring Security来锁定Actuator，但是因为Actuator本身没有安全保护，所以大多数端点默认都是禁用的，需要我们来选择对外暴露哪些

端点。

有两个配置属性能够控制对外暴露哪些端点，它们分别是management.endpoints.web.exposure.include和management.endpoints.web.exposure.exclude。通过management.endpoints.web. exposure.include属性，我们可以指定哪些端点想要暴露出来。例如，想要暴露“/health”“/info”“/beans”和“/conditions”的话，我们可以通过如下的配置来声明：

```
management:
  endpoints:
    web:
      exposure:
        include: health,info,beans,conditions
```

management.endpoints.web.exposure.include属性也可以接受星号（*）作为通配符，表明所有的Actuator端点都会对外暴露：

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

如果除了个别端点之外，我们想暴露其他的所有端点，那么一般来讲更简单的方式是通过通配符将它们全部包含进来，然后明确排除一部分。例如，我们想要暴露除了“/threaddump”和“/heapdump”之外的端点，那么可以按照如下的形式同时设置management.endpoints.web.exposure.include和

management.endpoints.web.exposure.exclude属性：

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
        exclude: threaddump,heapdump
```

如果你决定对外公开比“/health”和“/info”更多的信息，那么最好配置Spring Security来限制对其他端点的访问。我们将会在16.4节中了解如何保护Actuator端点。现在，我们看一下如何消费Actuator对外暴露的HTTP端点。

16.2　消费Actuator端点

Actuator是一个真正的宝藏，我们可以通过表16.1列出的HTTP端点获取正在运行中的应用的有用信息。作为HTTP端点，它们可以像任意REST API那样被消费，我们可以选择任意的HTTP客户端，包括Spring的RestTemplate和WebClient、基于浏览器的JavaScript应用以及简单的curl命令行客户端。

在探索Actuator端点的过程中，我们将会在本章中使用curl命令行客户端。在第17章中，我将会为你介绍Spring Boot Admin，这是一个构建在Actuator端点之上的用户友好的Web应用。

为了了解Actuator都提供了哪些端点，我们可以向Actuator的基础路径发送一个GET请求，这样能够得到每个端点的HATEOAS链接。如果

使用curl向“/actuator”发送一个请求，那么我们将会看到如下所示的响应（为了节省空间，进行了删减）：

```
$ curl localhost:8081/actuator
{
  "_links": {
    "self": {
      "href": "http://localhost:8081/actuator",
      "templated": false
    },
    "auditevents": {
      "href": "http://localhost:8081/actuator/auditevents",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8081/actuator/beans",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8081/actuator/health",
      "templated": false
    },
    ...
  }
}
```

因为不同的库可能会贡献自己的Actuator端点而且某些端点可能没有对外暴露，所以不同应用之间的实际结果也许会有所差异。

不管在什么情况下，Actuator基础路径提供的链接集合都可以作为Actuator所提供端点的一幅地图。我们首先从两个提供应用基本信息的端点开始探索Actuator，这两个端点就是“/health”和“/info”。

16.2.1　获取应用的基础信息

在去医院看病的时候，医生通常会首先问两个问题：你是谁？你感

觉怎样？尽管医生或护士选择的说法可能会有所不同，但是他们的最终目的都是想要了解接诊的人以及为什么要去医院找医生看病。

Actuator的“/info”和“/health”端点为Spring Boot应用同等重要的问题提供了答案。“/info”端点告诉我们关于应用的信息，而“/health”端点则告诉我们应用健康状况的信息。

请求关于应用的信息

要了解正在运行中的应用的信息，我们可以请求“/info”端点。但是，默认情况下，“/info”并不会提供什么信息。如下是我们使用curl发送请求后可能会看到的效果：

```
$ curl localhost:8081/actuator/info
{}
```

虽然这样看起来，“/info”端点似乎没有太大的用处，但是我们最好将它视为一块干净的画布，我们可以在上面绘制任何想要展现的信息。

我们可以有多种为“/info”端点提供信息的方式，但是最简单直接的就是创建一个或多个属性名带有“info.”前缀的配置属性。例如，假设我们希望在“/info”的响应中包含售后支持的联系信息，包括Email地址和电话号码。为了实现这一点，我们可以在application.yml文件中配置如下的属性：

```
info:
  contact:
    email: support@tacocloud.com
    phone: 822-625-6831
```

对于Spring Boot和应用上下文中的bean来说，info.contact.email property和info.contact.phone属性可能都没有什么特殊的意义。但是，因为它们的前缀是info，所以“/info”端点将会在响应中包含这两个属性的值：

```
{
  "contact": {
    "email": "support@tacocloud.com",
    "phone": "822-625-6831"
  }
}
```

在16.3.1小节，我们将会看到使用关于应用的有用信息来填充“/info”端点的其他几种方式。

探查应用的健康状况

发送HTTP GET请求到“/health”端点将会得到一个简单的JSON响应，其中包含了应用的健康状态。例如，如下是我们使用curl访问“/health”端点可能看到的响应：

```
$ curl localhost:8080/actuator/health
{"status":"UP"}
```

你可能会想，一个端点报告应用的状态是UP，这能有什么用处呢。如果应用停掉，那么它又该报告什么呢？

实际上，这里显示的是一个或多个健康指示器的聚合状态。健康指示器会报告应用要与之交互的外部系统的健康状态，比如数据库、消息代理甚至Spring Cloud组件，比如Eureka和Config Server。每个指示器的

健康状态可能会是如下的可选值中的某一个。

- UP：外部系统已经启动并且可以访问。
- DOWN：外部系统已经停机或者不可访问。
- UNKNOWN：外部系统的状态尚不清楚。
- OUT_OF_SERVICE：外部系统可以访问得到，但是目前不可用。

所有健康指示器的状态会聚合成应用整体的健康状态，这个过程中会使用如下的规则。

- 如果所有指示器都是UP，那么应用的健康状态是UP。
- 如果一个或多个健康指示器是DOWN，那么应用的健康状态就是DOWN。
- 如果一个或多个健康指示器是OUT_OF_SERVICE，那么应用的健康状态就是OUT_OF_SERVICE。
- UNKNOWN的健康状态会被忽略，不会计入应用的聚合状态中。

默认情况下，请求“/health”端点的响应中只会包含聚合的状态。但是，我们可以配置management.endpoint.health.show-details属性，以便于展示所有健康指示器的完整细节：

```
management:
  endpoint:
    health:
      show-details: always
```

management.endpoint.health.show-details属性的默认值是never。我们可以将它设置成always，这样就会始终显示健康指示器的完整细节；也可以将其设置成when-authorized，只有当客户端是完整认证的情况下才展示完整的细节信息。

现在，我们向“/health”端点发送GET请求的话，就会得到健康指示器的完整细节。如下是一个与Mongo文档数据库集成的服务样例：

```
{
  "status": "UP",
  "details": {
    "mongo": {
      "status": "UP",
      "details": {
        "version": "3.2.2"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 499963170816,
        "free": 177284784128,
        "threshold": 10485760
      }
    }
  }
}
```

所有的应用，不管其外部依赖是什么，至少都会有一个针对文件系统的健康指示器，名为diskSpace。diskSpace健康指示器能够显示文件系统的健康状况（希望它是UP状态），这个状态的值是由还有多少剩余空间决定的。如果可用磁盘空间低于阈值，那么它将会报告DOWN的状态。

在前面的样例中，还有一个mongo健康指示器，它报告了Mongo数据库的状态。细节信息包括了Mongo数据库的版本。

自动配置功能能够确保只有与应用程序相关的健康指示器才会显示到“/health”端点中。除了mongo和diskSpace健康指示器，Spring Boot还

为多个外部数据库和系统提供了健康指示器，包括：

- Cassandra
- Config Server
- Couchbase
- Eureka
- Hystrix
- JDBC数据源
- Elasticsearch
- InfluxDB
- JMS消息代理
- LDAP
- Email服务器
- Neo4j
- Rabbit消息代理
- Redis
- Solr

另外，第三方库可以贡献自己的健康指示器。我们将会在16.3.2小节看一下如何编写自定义的健康指示器。

我们可以看到“/info”和“/health”端点提供了正在运行中的应用的基本信息。同时，还有一些其他的Actuator端点能够探查应用内部的配置信息。接下来，我们看一下Actuator是如何展现应用的配置的。

16.2.2 查看配置细节

除了接收应用的基本信息之外，了解应用是如何配置的也很有指导意义。例如，应用上下文中都有哪些bean？自动配置中哪些条件通过了，哪些条件失败了？应用中有哪些可用的环境变量？HTTP请求是如何映射控制器的？某些包或类所设置的日志级别是什么？

这些问题可以通过Actuator的“/beans”“/conditions”“/env”“/configprops”“/mappings”和“/loggers”端点来回答。在有些情况下，我们甚至还可以使用“/env”和“/loggers”端点，在应用运行的过程中对配置信息进行调整。我们将会逐个看一下这些端点，它们能够让我们洞察正在运行中的应用的配置情况。下面首先从“/beans”端点开始。

获取**bean**的装配报告

要研究Spring应用上下文，最基础的端点就是“/beans”。这个端点返回的JSON文档描述了应用上下文中的每个bean，其中包括它的Java类型以及它被注入的其他bean。

对“/beans”端点发送GET请求的完整响应可以轻松地填满这一整章。所以，我们不会列出“/beans”的完整响应，而是只考虑下面的片段，主要关注一个bean条目：

```
{
  "contexts": {
    "application-1": {
      "beans": {
...
        "ingredientsController": {
          "aliases": [],
```

```
          "scope": "singleton",
          "type": "tacos.ingredients.IngredientsController",
          "resource": "file [/Users/habuma/Documents/Workspaces/
            TacoCloud/ingredient-service/target/classes/tacos/
            ingredients/IngredientsController.class]",
          "dependencies": [
            "ingredientRepository"
          ]
        },
...
      },
      "parentId": null
    }
  }
}
```

响应的根元素是contexts，它包含了一个子元素，代表应用中的每个Spring应用上下文。在每个应用上下文中，都有一个beans元素，它包含了应用上下文所有bean的细节。

在上面的样例中，显示了名为ingredientsController的bean。我们可以看到，它没有别名，scope是singleton并且类型为tacos.ingredients.IngredientsController。另外，resource属性指向了定义这个bean的类文件路径。dependencies属性列出了注入到给定bean的所有其他bean。在本例中，ingredientsController被注入了一个名为ingredientRepository的bean。

阐述自动装配

我们可以看到，自动装配是Spring Boot提供的最强大的功能之一。但是，有时候你可能想要知道这些功能为什么会自动装配在一起。或者，你认为某些功能已经自动装配了，但是它们实际上却没有，你可能

想要知道原因所在。在这种情况下，我们可以向“/conditions”发送GET请求，这样我们会知道自动装配过程中都发生了什么。

“/conditions”端点的自动装配报告可以分为3部分：匹配上的（positive matches，即已通过的条件化配置）、未匹配上的（negative matches，即失败的条件化配置）以及非条件化的类。如下的片段是对“/conditions”请求的响应，展现了每个组成部分的示例：

```
{
  "contexts": {
    "application-1": {
      "positiveMatches": {
...
        "MongoDataAutoConfiguration#mongoTemplate": [
          {
            "condition": "OnBeanCondition",
            "message": "@ConditionalOnMissingBean (types:
                org.springframework.data.mongodb.core.MongoTemplate;
                SearchStrategy: all) did not find any beans"
          }
        ],
...
      },
      "negativeMatches": {
...
        "DispatcherServletAutoConfiguration": {
          "notMatched": [
            {
              "condition": "OnClassCondition",
              "message": "@ConditionalOnClass did not find required
                  class 'org.springframework.web.servlet.
                                                   DispatcherServlet'"
            }
          ],
          "matched": []
        },
...
      },
      "unconditionalClasses": [
...
        "org.springframework.boot.autoconfigure.context.
```

```
                         ConfigurationPropertiesAutoConfiguration",
...
      ]
    }
  }
}
```

在positiveMatches区域中，我们可以看到通过自动配置创建了一个MongoTemplate bean，这是因为目前上下文中还没有这样的bean。导致这种配置结果的原因是这里包含了@ConditionalOnMissingBean注解，如果没有明确配置这个bean，就会自动配置它。在本例中，并没有找到MongoTemplate类型的bean，因此自动配置功能介入并创建了一个该类型的bean。

在negativeMatches区域中，Spring Boot要尝试配置一个DispatcherServlet。但是，@ConditionalOnClass条件化注解失败了，这是因为没有找到DispatcherServlet类。

最后，在unconditionalClasses区域中是一个无条件配置的ConfigurationPropertiesAutoConfiguration。配置属性是Spring Boot操作的基础，所以任何与配置属性相关的配置都应该无条件自动装配。

探查环境和配置属性

除了知道应用的bean是如何装配在一起的，我们可能还对有哪些可用的环境属性以及bean中都注入了哪些配置属性感兴趣。

当我们向“/env”端点发送GET请求的时候，我们会得到一个非常长的响应，它包含了Spring应用中所有发挥作用的属性源。其中包括来自

环境变量、JVM系统属性、application.properties和application.yml文件甚至来自Spring Cloud Config Server（该应用是Config Server客户端）的属性。

程序清单16.1列出了“/env”端点能够得到的响应示例，不过进行了删减，这样你会对它所提供的信息有一个大致了解：

程序清单16.1　“/env”端点的结果

```
$ curl localhost:8081/actuator/env
{
  "activeProfiles": [
    "development"
  ],
  "propertySources": [
...
    {
      "name": "systemEnvironment",
      "properties": {
        "PATH": {
          "value": "/usr/bin:/bin:/usr/sbin:/sbin",
          "origin": "System Environment Property \"PATH\""
        },
...
        "HOME": {
          "value": "/Users/habuma",
          "origin": "System Environment Property \"HOME\""
        }
      }
    },
    {
      "name": "applicationConfig: [classpath:/application.yml]",
      "properties": {
        "spring.application.name": {
          "value": "ingredient-service",
          "origin": "class path resource [application.yml]:3:11"
        },
        "server.port": {
          "value": 8081,
          "origin": "class path resource [application.yml]:9:9"
        },
```

```
...
      }
    },
...
  ]
}
```

“/env”的完整响应会包含更多的信息，但是程序清单 16.1只包含了几个值得注意的元素。首先，在响应的顶部是名为activeProfiles的字段。在本例中，它表明development profile处于激活状态。如果还有其他profile处于激活状态，那么也将会列到这里。

随后，propertySources字段是一个数组，Spring应用环境的每个属性源对应其中的一个条目。在程序清单16.1中，只显示了systemEnvironment以及引用application.yml文件的属性源。

在每个属性源中，是该属性源所提供的属性的列表以及它们的值。在application.yml属性源中，每个属性的origin字段指明了该属性是在哪里设置的，包括在application.yml文件中的行号和列号。

“/env”端点也可以用来获取特定的属性，只需要将属性名作为路径的第二个元素即可。例如，要检查server.port属性，我们可以提交GET请求到“/env/server.port”，如下所示：

```
$ curl localhost:8081/actuator/env/server.port
{
  "property": {
    "source": "systemEnvironment", "value": "8081"
  },
  "activeProfiles": [ "development" ],
  "propertySources": [
    { "name": "server.ports" },
    { "name": "mongo.ports" },
```

```
    { "name": "systemProperties" },
    { "name": "systemEnvironment",
      "property": {
        "value": "8081",
        "origin": "System Environment Property \"SERVER_PORT\""
      }
    },
    { "name": "random" },
    { "name": "applicationConfig: [classpath:/application.yml]",
      "property": {
        "value": 0,
        "origin": "class path resource [application.yml]:9:9"
      }
    },
    { "name": "springCloudClientHostInfo" },
    { "name": "refresh" },
    { "name": "defaultProperties" },
    { "name": "Management Server" }
  ]
}
```

我们可以看到，这里依然会展现所有的属性源，但是只有包含特定属性的属性源才会显示额外的信息。在本例中，systemEnvironment属性源和application.yml属性源都包含了server.port属性的值。因为systemEnvironment属性源要优先于后面所列的属性源，所以它的值8081会胜出。胜出的值也会反映在顶部的property字段中。

不仅可以用“/env”端点来读取属性的值，还可以通过向“/env”端点发送POST请求，同时提交JSON文档格式的name和value字段，为正在运行的应用设置属性。例如，要将名为tacocloud.discount.code的属性设置为TACOS1234，我们可以在命令行使用curl提交POST请求，如下所示：

```
$ curl localhost:8081/actuator/env \
       -d'{"name":"tacocloud.discount.code","value":"TACOS1234"}' \
       -H "Content-type: application/json"
{"tacocloud.discount.code":"TACOS1234"}
```

在提交该属性之后，在返回的响应中将会包含新设置的属性和它的值。如果后续不需要这个属性，那么我们可以提交一个DELETE请求到“/env”端点，将通过该端点创建的所有属性删除：

```
$ curl localhost:8081/actuator/env -X DELETE
{"tacocloud.discount.code":"TACOS1234"}
```

通过Actuator API设置属性是非常有用的，但是需要记住所有通过向“/env”端点发送POST请求设置的属性只会用到接收到该请求的应用中，是临时的，应用重启的话就会丢失。

HTTP映射导览

尽管Spring MVC（以及Spring WebFlux）编程模型非常易于处理HTTP请求，我们只需要为方法添加请求映射注解即可，但是我们很难对应用整体能够处理哪些HTTP请求以及每种组件分别处理哪些请求有一个整体的了解。

Actuator的“/mappings”端点为应用中的所有HTTP请求处理器提供了一个一站式的视图，不管这些处理器是来自Spring MVC控制器还是Actuator端点，我们都能一目了然地看清。要获取Spring Boot应用中所有端点的完整列表，我们只需要向“/mappings”发送一个GET请求，就会看到大致如程序清单16.2所示的响应。

程序清单16.2　“/mappings”端点所展示的HTTP映射

```
$ curl localhost:8081/actuator/mappings | jq
```

```
{
  "contexts": {
    "application-1": {
      "mappings": {
        "dispatcherHandlers": {
          "webHandler": [
...
            {
              "predicate": "{[/ingredients],methods=[GET]}",
              "handler": "public
      reactor.core.publisher.Flux<tacos.ingredients.Ingredient>
      tacos.ingredients.IngredientsController.allIngredients()",
              "details": {
                "handlerMethod": {
                  "className": "tacos.ingredients.IngredientsController",
                  "name": "allIngredients",
                  "descriptor": "()Lreactor/core/publisher/Flux;"
                },
                "handlerFunction": null,
                "requestMappingConditions": {
                  "consumes": [],
                  "headers": [],
                  "methods": [
                    "GET"
                  ],
                  "params": [],
                  "patterns": [
                    "/ingredients"
                  ],
                  "produces": []
                }
              }
            },
...
          ]
        }
      },
      "parentId": "application-1"
    },
    "bootstrap": {
      "mappings": {
        "dispatcherHandlers": {}
      },
      "parentId": null
    }
  }
}
```

为了简洁，这个响应进行了删减，只包含了一个请求处理器。具体来讲，它表明对“/ingredients”的GET请求将由IngredientsController的allIngredients()方法来处理。

管理日志级别

对于任何应用来说，日志都是很重要的特性。日志是一种审计方式，也是一种较为粗略的调试方法。

设置日志级别是一种需要很强平衡能力的事情。如果我们将日志级别设置得太低，那么日志中会有太多的噪声，查找有用的信息会变得很困难。另外，如果我们将日志级别设置地过于简洁，那么日志对于理解应用正在做什么可能没有太大的价值。

日志级别通常会基于Java包来进行设置。如果你想要知道正在运行的应用中使用了什么日志级别，那么可以向“/loggers”端点发送GET请求。如下的JSON展示了“/loggers”响应的一个片段：

```
{
  "levels": [ "OFF", "ERROR", "WARN", "INFO", "DEBUG", "TRACE" ],
  "loggers": {
    "ROOT": {
      "configuredLevel": "INFO", "effectiveLevel": "INFO"
    },
...
    "org.springframework.web": {
      "configuredLevel": null, "effectiveLevel": "INFO"
    },
...
    "tacos": {
      "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients": {
```

```
      "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients.IngredientServiceApplication": {
      "configuredLevel": null, "effectiveLevel": "INFO"
    }
  }
}
```

在响应的顶部首先是所有合法日志级别的列表。在此之后，loggers元素列出了应用中每个包的日志级别详情。configuredLevel属性展示了明确配置的日志级别（如果没有明确配置的话，将会显示null）。effectiveLevel属性展示的是实际的日志级别，它可能是从父包或根logger继承下来的。

尽管这个片段只展现了根logger和4个包的日志级别，但是完整的响应会包含应用中每个包的日志级别，包括我们所使用的库对应的包。如果你只关心特定的包，那么可以在请求中以额外路径组件的方式指明包的名称。

例如，你只想知道taco.ingredients包的日志级别，那么可以发送请求到“/loggers/tacos/ ingredients”：

```
{
  "configuredLevel": null,
  "effectiveLevel": "INFO"
}
```

除了返回应用程序中包的日志级别之外，通过向“/loggers”端点发送POST请求，我们还能修改已配置的日志级别。例如，假设我们想要将taco.ingredients包的日志级别设置为DEBUG。如下的curl命令能够实现这一点：

```
$ curl localhost:8081/actuator/loggers/tacos/ingredients \
       -d'{"configuredLevel":"DEBUG"}' \
       -H"Content-type: application/json"
```

现在，日志级别已经发生了变化，我们可以向"/loggers/tacos/ingredients"发送GET请求，看一下它变成了什么样子：

```
{
  "configuredLevel": "DEBUG",
  "effectiveLevel": "DEBUG"
}
```

注意，在此之前，configuredLevel的值为null，现在它变成了DEBUG。这个变更也会影响到effectiveLevel。最重要的是，如果这个包中的代码以debug级别打印日志，那么日志文件中将会包含debug级别的信息。

16.2.3 查看应用的活动

如果我们能够时刻监视运行中应用的活动，那将会非常有用，我们所关注的信息可能包括应用正在处理什么类型的HTTP请求以及应用中所有线程的活动。为了实现这一点，Actuator提供了"/httptrace""/threaddump"和"/heapdump"端点。

"/heapdump"端点可能是最难以详细阐述的Actuator端点。简而言之，它会下载一个gzip压缩的HPROF堆转储文件，该文件可以用来跟踪内存和线程问题。由于篇幅的原因，再加上堆转储文件的使用是一个非常高级的特性，所以对"/heapdump"端点的介绍就仅限于此。

跟踪**HTTP**活动

“/httptrace”端点能够报告应用所处理的最近100个请求的详情。详情内容包括请求的方法和路径、代表请求处理时刻的时间戳、请求和响应的头信息以及处理该请求的耗时。

如下的JSON片段展示了“/httptrace”端点响应的一个条目：

```
{
  "traces": [
    {
      "timestamp": "2018-06-03T23:41:24.494Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8081/ingredients",
        "headers": {
          "Host": ["localhost:8081"],
          "User-Agent": ["curl/7.54.0"],
          "Accept": ["*/*"]
        },
        "remoteAddress": null
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": ["application/json;charset=UTF-8"]
        }
      },
      "timeTaken": 4
    },
...
  ]
}
```

尽管这些信息对调试很有价值，但是随着时间推移不断跟踪数据是更有意思的，基于响应的状态值，它能够让我们洞察应用程序在给定的

时间内有多少请求是成功的、有多少请求是失败的。在第17章中，我们将会看到Spring Boot Admin是如何将这些信息捕获到一个运行图中的，借助这个图我们能够可视化一定的时间范围内的HTTP跟踪信息。

监控线程

除了HTTP请求的跟踪信息，在确定应用运行状况的时候，线程活动也是非常有用的。“/threaddump”端点能够生成一个当前线程活动的快照。通过如下的“/threaddump”端点响应片段，我们能够大致了解这个端点都提供了什么功能：

```
{
  "threadName": "reactor-http-nio-8",
  "threadId": 338,
  "blockedTime": -1,
  "blockedCount": 0,
  "waitedTime": -1,
  "waitedCount": 0,
  "lockName": null,
  "lockOwnerId": -1,
  "lockOwnerName": null,
  "inNative": true,
  "suspended": false,
  "threadState": "RUNNABLE",
  "stackTrace": [
    {
      "methodName": "kevent0",
      "fileName": "KQueueArrayWrapper.java",
      "lineNumber": -2,
      "className": "sun.nio.ch.KQueueArrayWrapper",
      "nativeMethod": true
    },
    {
      "methodName": "poll",
      "fileName": "KQueueArrayWrapper.java",
      "lineNumber": 198,
      "className": "sun.nio.ch.KQueueArrayWrapper",
      "nativeMethod": false
```

```
    },
...
  ],
  "lockedMonitors": [
    {
      "className": "io.netty.channel.nio.SelectedSelectionKeySet",
      "identityHashCode": 1039768944,
      "lockedStackDepth": 3,
      "lockedStackFrame": {
        "methodName": "lockAndDoSelect",
        "fileName": "SelectorImpl.java",
        "lineNumber": 86,
        "className": "sun.nio.ch.SelectorImpl",
        "nativeMethod": false
      }
    },
...
  ],
  "lockedSynchronizers": [],
  "lockInfo": null
}
```

完整的线程转储报告包含了运行中应用的每个线程。为了节省空间，这里的线程转储进行了删减，只包含了一个线程条目。我们可以看到，这里包含了线程的阻塞和锁定状态，以及其他的线程细节。这里还有一个堆栈，它能够展现线程都将时间花到了哪块代码中。

因为“/threaddump”只提供了请求时线程活动的快照，所以它很难完整了解随着时间的推移线程的行为都是什么样子的。在第17章中，我们将会看到Spring Boot Admin如何在一个实时视图中监视“/threaddump”端点。

16.2.4 获取应用的指标

“/metrics”端点能够报告运行中的应用程序所生成的各种度量指标，

包括关于内存、处理器、垃圾收集以及HTTP请求的指标。Actuator提供了20多个开箱即用的指标分类，当我们向“/metrics”发送GET请求时所得到的指标分类证明了这一点：

```
$ curl localhost:8081/actuator/metrics | jq
{
  "names": [
    "jvm.memory.max",
    "process.files.max",
    "jvm.gc.memory.promoted",
    "http.server.requests",
    "system.load.average.1m",
    "jvm.memory.used",
    "jvm.gc.max.data.size",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "jvm.buffer.memory.used",
    "jvm.threads.daemon",
    "system.cpu.usage",
    "jvm.gc.memory.allocated",
    "jvm.threads.live",
    "jvm.threads.peak",
    "process.uptime",
    "process.cpu.usage",
    "jvm.classes.loaded",
    "jvm.gc.pause",
    "jvm.classes.unloaded",
    "jvm.gc.live.data.size",
    "process.files.open",
    "jvm.buffer.count",
    "jvm.buffer.total.capacity",
    "process.start.time"
  ]
}
```

这里涉及太多的指标，所以本章不可能面面俱到地介绍。相反，我们可以只关注一个指标分类，即http.server.requests，以它作为样例介绍如何消费“/metrics”端点。

现在，我们不再简单地请求“/metrics”，而是发送GET请求到“/metrics/{METRICS CATEGORY}”，这样我们就会收到该分类的指标详情。就http.server.requests来说，我们发送GET请求到“/metrics/http.server.requests”所返回的数据如下所示：

```
$ curl localhost:8081/actuator/metrics/http.server.requests
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 2103 },
    { "statistic": "TOTAL_TIME", "value": 18.086334315 },
    { "statistic": "MAX", "value": 0.028926313 }
  ],
  "availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException",
                  "IllegalArgumentException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
        "/actuator/metrics/{requiredMetricName}",
        "/actuator/health", "/actuator/info", "/ingredients",
        "/actuator/metrics", "/**" ] },
    { "tag": "status", "values": [ "404", "500", "200" ] }
  ]
}
```

这个响应中最重要的组成部分是measurements区域，它包含了所请求分类的所有指标数据。在本例中，它表示一共有2103个HTTP请求，处理这些请求的总耗时是18.086334315秒，处理单个请求的最大耗时是0.028926313秒。

这些通用的指标非常有意思，但是我们可以使用availableTags中所列出的标签进一步细化结果。例如，我们知道一共有2103个请求，但是还不知道HTTP 200、HTTP 404或HTTP 500响应状态的请求分别有多

少。借助status标签，我们可以得到所有状态为HTTP 404的请求指标：

```
$ curl localhost:8081/actuator/metrics/http.server.requests? \
                                       tag=status:404
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 31 },
    { "statistic": "TOTAL_TIME", "value": 0.522061212 },
    { "statistic": "MAX", "value": 0 }
  ],
  "availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
        "/actuator/metrics/{requiredMetricName}", "/**" ] }
  ]
}
```

通过使用tag请求属性指定标签名和值，我们可以看到所有响应为HTTP 404的请求的指标。这里显示有31个请求的结果是404，耗用了0.522061212秒。除此之外，我们可以看到有一些失败的请求是针对“/actuator/metrics/{requiredMetricsName}”的GET请求（尽管我们并不清楚{requiredMetricsName}路径变量解析成了什么）。另外，有些是发送其他路径的，是由“/**”通配符捕获到的。

如果我们想要知道有多少HTTP 404响应是发送到“/**”路径的，那么又该怎么办呢？我们所要做的就是进一步对其进行过滤，在请求中使用url标签，如下所示：

```
% curl "localhost:8081/actuator/metrics/http.server.requests? \
                                    tag=status:404&tag=uri:/**"
{
  "name": "http.server.requests",
```

```
  "measurements": [
    { "statistic": "COUNT", "value": 30 },
    { "statistic": "TOTAL_TIME", "value": 0.519791548 },
    { "statistic": "MAX", "value": 0 }
  ],
  "availableTags": [
    { "tag": "exception", "values": [ "ResponseStatusException" ] },
    { "tag": "method", "values": [ "GET" ] }
  ]
}
```

我们可以看到有30个路径匹配“/**”的请求得到了HTTP 404，并且处理这些请求耗费了0.519791548秒。

你可能也注意到了，随着我们不断细化请求的条件，可用的标签越来越有限。这里只列出了展现指标所对应的请求能够适用的标签。在本例中，exception和method标签只有一个值。显然，30个请求都是GET请求，并且都是因为抛出ResponseStatusException而产生的404状态。

导览整个“/metrics”可能是一件很麻烦的事情，但是稍加练习，我们一定能够找到自己想要的数据。在第17章中，我们将会看到借助Spring Boot Admin，我们能够更容易地消费“/metrics”端点的数据。

尽管Actuator端点所提供的信息有助于观察运行中Spring Boot应用的内部状况，但是它们并不适用于人类直接使用。作为REST端点，它们是供其他应用消费的，这里所说的其他应用也可能是UI。考虑到这一点，我们在第17 章会看到如何在用户友好的Web应用中展现Actuator信息。现在，我们看一下如何自定义Actuator的端点。

16.3　自定义Actuator

Actuator最棒的特性之一就是它能够进行自定义，以满足应用的特定需求。有一些端点本身支持自定义扩展，同时Actuator也允许我们创建完全自定义的端点。

接下来，我们看一下Actuator能够进行自定义的几种方式。下面先从为“/info”端点添加信息开始。

16.3.1　为“/info”端点提供信息

正如我们在16.2.1小节所看到的那样，“/info”最初是空的，没有提供任何信息。我们可以通过创建前缀为“info.”的属性很容易地为它添加数据。

尽管创建前缀为“info.”的属性是一个很简单的为“/info”端点添加自定义数据的方式，但是这并不是唯一的方式。Spring Boot提供了名为InfoContributor的接口，允许我们以编程的方式为“/info”端点添加任何想要的信息。Spring Boot甚至提供了InfoContributor接口的几个实现，你肯定会发现它们非常有用。

接下来，我们看一下如何编写自定义的InfoContributor，以便于向“/info”端点添加自定义的信息。

创建自定义的**Info**贡献者

假设我们想要为“/info”端点添加关于Taco Cloud的统计信息，比如

想要包含已经创建多少taco的信息。为了实现这一点，我们需要编写一个实现InfoContributor接口的类，并将TacoRepository注入进来，然后发布TacoRepository提供的信息到“/info”端点中。程序清单16.3展示了如何实现这样一个贡献者。

程序清单16.3　InfoContributor的自定义实现

```
package tacos.tacos;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;
import org.springframework.boot.actuate.info.Info.Builder;

@Component
public class TacoCountInfoContributor implements InfoContributor {
  private TacoRepository tacoRepo;

  public TacoCountInfoContributor(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @Override
  public void contribute(Builder builder) {
    long tacoCount = tacoRepo.count();
    Map<String, Object> tacoMap = new HashMap<String, Object>();
    tacoMap.put("count", tacoCount);
    builder.withDetail("taco-stats", tacoMap);
  }
}
```

要实现InfoContributor接口，TacoCountInfoContributor就需要实现contribute()方法。这个方法能够得到一个Builder对象，基于这个对象，contribute()调用withDetail()方法来添加详情信息。在上述的实现中，我们通过TacoRepository的count()来获取已经创建了多少个taco。然后，我们将这个数量放到一个Map中，以值为taco-stats的label将它传递到

builder中。这样形成的“/info”端点将会包含这个数量，如下所示：

```
{
  "taco-stats": {
    "count": 44
  }
}
```

我们可以看到，InfoContributor的实现可以以任何方式贡献信息。为属性添加“info.”前缀虽然简单，但是它们却只能是静态值。

注入构建信息到**“/info”**端点中

Spring Boot提供了一些内置的InfoContributor实现，它们能够自动添加信息到“/info”端点的结果中。其中有一个实现是BuildInfoContributor，它能够将项目构建文件中的信息添加到“/info”端点的结果中。这包括了一些基本信息，比如项目版本、构建的时间戳以及执行构建的主机和用户。

为了将构建信息添加到“/info”端点的结果中，我们需要添加build-info goal到Spring Boot Maven Plugin executions中，如下所示：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
```

```
    </plugin>
  </plugins>
</build>
```

如果使用Gradle构建项目，我们只需要添加如下几行代码到build.gradle文件中：

```
springBoot {
  buildInfo()
}
```

不管是哪种方式，构建过程都会在可分发的JAR或WAR文件中生成一个名为build-info.properties的文件，BuildInfoContributor会使用这个文件并为“/info”端点贡献信息。如下的“/info”端点响应片段展现了所贡献的构建信息：

```
{
  "build": {
    "version": "0.0.16-SNAPSHOT",
    "artifact": "ingredient-service",
    "name": "ingredient-service",
    "group": "sia5",
    "time": "2018-06-04T00:24:04.373Z"
  }
}
```

这个信息对于我们理解正在运行的应用的确切版本和构建时间是非常有用的。通过向“/info”端点发送GET请求，我们就能知道正在运行的是不是项目的最新构建版本。

暴露**Git**提交信息

假设我们的项目使用Git进行源码控制，那么我们可以在“/info”端点

中包含Git提交信息。为了实现这一点，我们需要添加如下的插件到Maven项目的pom.xml文件中：

```
<build>
  <plugins>
...
    <plugin>
      <groupId>pl.project13.maven</groupId>
      <artifactId>git-commit-id-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

如果你是Gradle用户，也不用担心，我们可以将一个功能相同的插件放到build.gradle文件中：

```
plugins {
  id "com.gorylenko.gradle-git-properties" version "1.4.17"
}
```

这两个插件完成的是相同的事情：它们会生成一个名为git.properties的构建期制件，这个文件包含了项目的所有Git元数据。在运行时，有个特殊的InfoContributor实现能够发现这个文件并将它的内容贡献给“/info”端点。

按照最简单的形式，“/info”端点展现的Git信息包括应用构建所使用的Git分支、提交的哈希值以及时间戳：

```
{
  "git": {
    "commit": {
      "time": "2018-06-02T18:10:58Z",
      "id": "b5c104d"
    },
    "branch": "master"
```

```
  },
...
}
```

这些信息非常确定地描述了项目构建时代码的状态。但是，我们还可以将management.info.git.mode属性设置为full：

```
management:
  info:
    git:
      mode: full
```

这样我们就能得到项目构建时非常详尽的Git提交信息。程序清单16.4展现了完整Git信息的一个样例。

程序清单16.4　通过“/info”端点展现完整的Git信息

```
{
  "git": {
    "build": {
      "host": "DarkSide.local",
      "version": "0.0.16-SNAPSHOT",
      "time": "2018-06-02T18:11:23Z",
      "user": {
        "name": "Craig Walls",
        "email": "craig@habuma.com"
      }
    },
    "branch": "master",
    "commit": {
      "message": {
        "short": "Add Spring Boot Admin and Actuator",
        "full": "Add Spring Boot Admin and Actuator"
      },
      "id": {
        "describe": "b5c104d-dirty",
        "abbrev": "b5c104d",
        "describe-short": "b5c104d-dirty",
        "full": "b5c104d1fcbe6c2b84965ea08a330595100fd44e"
      },
      "time": "2018-06-02T18:10:58Z",
```

```
      "user": {
        "email": "craig@habuma.com",
        "name": "Craig Walls"
      }
    },
    "closest": {
      "tag": {
        "name": "",
        "commit": {
          "count": ""
        }
      }
    },
    "dirty": "true",
    "remote": {
      "origin": {
        "url": "Unknown"
      }
    },
    "tags": ""
  },
...
}
```

除了时间戳和Git提交哈希值的缩略值，完整版本的信息还包含了代码提交者的名字和邮箱、完整的提交信息和其他内容，这样我们就能精确定位构建项目所使用的代码。实际上，我们可以看到程序清单16.4中dirty属性的值为true，表明在项目构建时构建目录中存在未提交的变更。没有什么信息比这更有说服力了！

16.3.2 实现自定义的健康指示器

Spring Boot提供了多个内置的健康指示器，它们能够提供与Spring应用进行交互的通用外部系统的健康信息。有时候你可能会发现，你所使用的外部系统在Spring Boot的预料之外，Spring Boot也没有为它提供

健康指示器。例如，你的应用可能与一个遗留的大型机应用进行交互，应用的健康状况可能会受到遗留系统健康状况的影响。为了创建自定义的健康指示器，我们需要做的就是创建一个实现了HealthIndicator接口的bean。

实际上，Taco Cloud服务没有必要创建自定义的健康指示器，Spring Boot所提供的指示器就足够用了。为了阐述如何开发自定义的健康指示器，我们看一下程序清单16.5。它展现了一个简单的HealthIndicator实现，健康状况由每天中的时间所决定。

程序清单16.5　HealthIndicator的一个特殊实现

```
package tacos.tacos;
import java.util.Calendar;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class WackoHealthIndicator
       implements HealthIndicator {
  @Override
  public Health health() {
    int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
    if (hour > 12) {
      return Health
          .outOfService()
          .withDetail("reason",
                 "I'm out of service after lunchtime")
          .withDetail("hour", hour)
          .build();
    }

    if (Math.random() < 0.1) {
      return Health
          .down()
          .withDetail("reason", "I break 10% of the time")
          .build();
```

```
    }

    return Health
        .up()
        .withDetail("reason", "All is good!")
        .build();
  }
}
```

这个疯狂的健康指示器首先会判断当前是什么时间。如果是下午，那么所返回的健康状态是OUT_OF_SERVICE，其中还包含导致该状态的原因详情。即便是在午饭前，这个健康指示器也有10%的概率报告DOWN状态，因为它使用随机数来决定应用是否正常启动。如果随机数的值小于0.1，那么状态将是DOWN，否则状态将是UP。

显然，在真正的应用中，程序清单16.5 的健康指示器不会有什么用处。但是，可以假设一下，我们不是根据当前时间或随机数，而是对外部系统发起一个远程调用，并基于接收到的响应状态来进行判定，这样的话它就是一个非常有用的健康指示器了。

16.3.3　注册自定义的指标

在16.2.4小节中，我们看到了如何访问“/metrics”端点来消费Actuator发布的各种指标，当时我们主要关注了HTTP请求的信息。Actuator提供的指标非常有用，但是“/metrics”端点的结果并不局限于内置的指标。

实际上，Actuator的指标是由Micrometer实现的。这是一个供应商中立的指标门面，借助它，我们能够发送任意想要的指标，并在所选的第三方监控系统中对其进行展现。它提供了对Prometheus、Datadog和

New Relic等系统的支持。

使用Micrometer发布指标的最基本方式是借助Micrometer的MeterRegistry。在Spring Boot应用中，要发布指标的话，我们唯一需要做的就是将MeterRegistry注入到想要发布计数器、计时器和计量器（gauges）的地方，这些地方能够捕获应用的指标信息。

作为发布自定义指标的样例，假设我们想要统计不同配料所创建的taco的数量。也就是说，我们想要知道，使用生菜、碎牛肉、墨西哥薄饼以及其他配料分别制作了多少个taco。程序清单16.6中的TacoMetrics bean展示了如何使用MeterRegistry来收集信息。

程序清单16.6　TacoMetrics注册了关于taco配料的指标

```
package tacos.tacos;
import java.util.List;
import
     org.springframework.data.rest.core.event.AbstractRepositoryEventListe
ner
     ;
import org.springframework.stereotype.Component;
import io.micrometer.core.instrument.MeterRegistry;

@Component
public class TacoMetrics extends AbstractRepositoryEventListener<Taco> {
  private MeterRegistry meterRegistry;
  public TacoMetrics(MeterRegistry meterRegistry) {
    this.meterRegistry = meterRegistry;
  }

  @Override
  protected void onAfterCreate(Taco taco) {
    List<Ingredient> ingredients = taco.getIngredients();
    for (Ingredient ingredient : ingredients) {
      meterRegistry.counter("tacocloud",
          "ingredient", ingredient.getId()).increment();
    }
```

```
  }
}
```

我们可以看到，TacoMetrics通过其构造器注入了MeterRegistry。它还扩展了AbstractRepositoryEventListener，这是Spring Data中的一个类，能够拦截repository事件。我们重写了onAfterCreate()方法，这样每当保存新的Taco对象时都会得到通知。

在onAfterCreate()中，我们为每种配料声明了一个计数器，其中标签名为ingredient，标签值为配料ID。如果给定标签的计数器已经存在，就会重用已有的计数器。计数器会不断递增，表明使用该配料又创建了一个taco。

在创建完几个taco之后，我们就可以查询“/metrics”端点来获取配料的计数信息了。对“/metrics/tacocloud”发送GET请求将会生成如下未经过滤的指标数据：

```
$ curl localhost:8087/actuator/metrics/tacocloud
{
  "name": "tacocloud",
  "measurements": [ { "statistic": "COUNT", "value": 84 }
  ],
  "availableTags": [
    {
      "tag": "ingredient",
      "values": [ "FLTO", "CHED", "LETC", "GRBF",
                  "COTO", "JACK", "TMTO", "SLSA"]
    }
  ]
}
```

measurements下的数值并没有太大的用处，它代表了所有配料的总数。但是，如果你想要知道使用墨西哥薄饼（FLTO）创建了多少个

taco，那么我们可以将ingredient标签的值设置为FLTO：

```
$ curl localhost:8087/actuator/metrics/tacocloud?tag=ingredient:FLTO

{
  "name": "tacocloud",
  "measurements": [
    { "statistic": "COUNT", "value": 39 }
  ],
  "availableTags": []
}
```

现在，我们可以清楚地看到，有39个taco是使用墨西哥薄饼作为其中的一道配料创建的。

16.3.4 创建自定义的端点

乍看上去，你可能会认为Actuator端点不过是使用Spring MVC的控制器实现的，但是在第18章中你将会发现，这些端点除了通过HTTP请求暴露之外，还暴露成JMX MBean。因此，它们肯定不仅仅是控制器类的端点。

实际上，Actuator端点的定义与控制器有很大的差异。Actuator端点并不是使用@Controller或@RestController注解来标注类，而是通过为类添加@Endpoint注解来实现的。

另外，它们不是使用HTTP方法命名的注解，如@GetMapping、@PostMapping或@DeleteMapping，Actuator端点的操作是通过为方法添加@ReadOperation、@WriteOperation和@DeleteOperation注解实现的。这些注解并没有指明任何的通信机制，实际上，这允许Actuator与各种

各样的通信机制协作，内置了对HTTP和JMX的支持。

为了阐述如何编写自定义的Actuator，参见程序清单16.7中的NotesEndpoint。

程序清单16.7　用来记笔记的自定义端点

```
package tacos.ingredients;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import org.springframework.boot.actuate.endpoint.annotation.DeleteOperatio
n;
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.boot.actuate.endpoint.annotation.WriteOperation
;
import org.springframework.stereotype.Component;
import lombok.Getter;
import lombok.RequiredArgsConstructor;

@Component
@Endpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {

  private List<Note> notes = new ArrayList<>();

  @ReadOperation
  public List<Note> notes() {
    return notes;
  }
  @WriteOperation
  public List<Note> addNote(String text) {
    notes.add(new Note(text));
    return notes;
  }

  @DeleteOperation
  public List<Note> deleteNote(int index) {
    if (index < notes.size()) {
      notes.remove(index);
    }
    return notes;
```

```
  }

  @RequiredArgsConstructor
  private class Note {
    @Getter
    private Date time = new Date();

    @Getter
    private final String text;
  }
}
```

这是一个非常简单的记笔记的端点，我们可以通过写入操作提交笔记，通过读取操作阅读笔记列表并且通过删除操作移除某个笔记。不得不承认，这个端点并不像Actuator的端点那样有用。但是考虑到开箱即用的Actuator端点提供了如此众多的功能，所以很难设想一个自定义Actuator端点的实际样例。

不管怎么说，NotesEndpoint类使用了@Component注解，这样它会被Spring的组件扫描所发现，并将其初始化为Spring应用上下文中的bean。但是，与我们的讨论关联最大的事情是它还使用了@Endpoint注解，使其成为一个ID为notes的Actuator端点。它默认就是启用的，所以我们不需要在management.web.endpoints.web.exposure.include配置属性中显式启用它。

可以看到，NotesEndpoint提供了各种类型的操作。

- notes()方法使用了@ReadOperation注解。当它被调用的时候，将会返回一个可用笔记的列表。按照HTTP的术语，这意味着它会处理针对“/actuator/notes”的HTTP GET请求并返回JSON格式的笔记列表。

- addNote()方法使用了@WriteOperation注解。当它被调用的时候，将会根据给定的文本创建一个新的笔记并添加到列表中。按照HTTP的术语，它处理POST请求，请求体中是一个包含text属性的JSON对象。最后，它会在响应中返回当前笔记列表的状态。
- deleteNote()方法使用了@DeleteOperation注解。当它被调用的时候，将会根据给定的索引删除一条笔记。按照HTTP的术语，这个端点会处理DELETE请求，其中索引是通过请求参数设置进来的。

为了看一下它的实际效果，我们可以使用curl测试新的端点。首先，使用两个单独的POST请求，添加两条笔记：

```
$ curl localhost:8080/actuator/notes \
               -d'{"text":"Bring home milk"}' \
               -H"Content-type: application/json"
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}]

$ curl localhost:8080/actuator/notes \
               -d'{"text":"Take dry cleaning"}' \
               -H"Content-type: application/json"
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"},
{"time":"2018-06-08T13:50:48.021+0000","text":"Take dry cleaning"}]
```

我们可以看到，每当新增笔记的时候，端点都会返回增加新内容之后的笔记列表。如果想要查看笔记列表，我们可以发送一个简单的GET请求：

```
$ curl localhost:8080/actuator/notes
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"},
 {"time":"2018-06-08T13:50:48.021+0000","text":"Take dry cleaning"}]
```

如果决定移除其中的某条笔记，那么我们可以发送一个DELETE请求，并将index作为请求参数：

```
$ curl localhost:8080/actuator/notes?index=1 -X DELETE
```

```
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}]
```

很重要的一点就是，尽管我只展现了如何使用HTTP与端点交互，但是它们还会暴露为MBean，我们可以使用任意的JMX客户端来进行访问。如果你只想暴露HTTP端点，那么可以使用@WebEndpoint注解而不是@Endpoint来标注端点类：

```
@Component
@WebEndpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {
  ...
}
```

类似的，如果你只想暴露MBean端点，那么可以使用@JmxEndpoint注解进行标注。

16.4 保护Actuator

我们可能不想让别人窥探Actuator暴露的信息。另外，因为Actuator提供了一些操作来修改环境变量和日志级别，所以最好对Actuator进行保护，只有具有对应权限的客户端才能消费这些端点。

虽然保护Actuator端点非常重要，但是安全性本身并不是Actuator的职责，我们需要使用Spring Security来保护Actuator。因为Actuator端点的路径和应用本身的路径非常相似，所以保护Actuator与保护其他的应用路径并没有什么区别。我们在第4章讨论的内容依然适用于保护Actuator端点。

因为所有的端点都集中在“/actuator”基础路径（如果设置了

management.endpoints.web.base-path属性，那么可能会是其他的路径）下，所以很容易将授权规则应用到所有的Actuator端点上。例如，只有具有ROLE_ADMIN权限的用户才能调用Actuator端点，那么我们可以重写WebSecurityConfigurerAdapter的configure()方法：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/actuator/**").hasRole("ADMIN")

    .and()

    .httpBasic();
}
```

这需要所有的请求均由具备ROLE_ADMIN权限的授权用户发起，才能进行访问。它还配置了HTTP basic认证，这样客户端应用可以在请求的Authorization头信息中提交编码后的认证信息。

保护Actuator的唯一问题在于，端点的路径硬编码为“/actuator/**”，如果因为修改了management.endpoints.web.base-path属性发生变化的话，那么这种方式就无法正常运行了。为了帮助解决这个问题，Spring Boot提供了EndpointRequest（一个请求匹配类，更简单，而且不依赖于给定的String路径）。借助EndpointRequest，我们可以将相同的安全要求用到Actuator上，而且不需要硬编码路径：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .requestMatcher(EndpointRequest.toAnyEndpoint())
      .authorizeRequests()
        .anyRequest().hasRole("ADMIN")
```

```
    .and()
    .httpBasic();
}
```

EndpointRequest.toAnyEndpoint()方法会返回一个请求匹配器，它会匹配所有的Actuator端点。如果你想要将某些端点从请求匹配器中移除，那么我们可以调用excluding()方法，通过名称进行声明：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .requestMatcher(
        EndpointRequest.toAnyEndpoint()
                       .excluding("health", "info"))
    .authorizeRequests()
      .anyRequest().hasRole("ADMIN")
  .and()
    .httpBasic();
}
```

另外，如果我们只是想将安全性用到其中一部分Actuator端点中，那么可以调用to()来替换toAnyEndpoint()，并使用名称指明这些端点：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .requestMatcher(EndpointRequest.to(
            "beans", "threaddump", "loggers"))
    .authorizeRequests()
      .anyRequest().hasRole("ADMIN")
  .and()
    .httpBasic();
}
```

这样会限制只将安全性功能用到“/beans”“/threaddump”和“/loggers”端点上，其他的Actuator端点会全部对外开放。

16.5 小结

- Spring Boot Actuator以HTTP和JMX MBean的形式提供了多个端点，它们能够让我们探查Spring Boot应用内部的运行状况。
- 大多数的Actuator端点默认是禁用的，我们可以通过设置management.endpoints.web.exposure.include和management.endpoints.web.exposure.exclude属性有选择地对外暴露。
- 有些端点，比如"/loggers"和"/env"，允许写入操作，这样能够在运行时改变应用的配置。
- 借助"/info"端点可以暴露应用的构建和Git提交的详情。
- 自定义的健康指示器可以反映应用的健康状况，以便于跟踪外部集成系统的健康状态。
- 自定义的应用指标可以通过Micrometer进行注册，让Spring Boot应用与多种流行的指标引擎进行集成，包括Datadog、New Relic和Prometheus。
- Actuator的Web端点可以通过Spring Security进行保护，与Spring Web应用的其他端点非常相似。

第17章　管理Spring

本章内容：

- 搭建Spring Boot Admin
- 注册客户端应用
- 使用Actuator端点
- 保护Admin服务器

“一图胜千言”，对于很多应用程序的用户来说，一个用户友好的Web应用要胜过上千个API调用。不要误会我的意思，我是一个命令行爱好者，非常喜欢使用curl和HTTPie消费REST API。但是有时候先手动输入命令行来调用REST端点再查看结果要比在浏览器中点击链接并阅读结果低效得多。

在前面的章节中，我们探索了Spring Boot Actuator暴露的所有HTTP端点。端点返回的是JSON响应，所以对于如何使用它们并没有任何限制。在本章中，我们将会看到基于Actuator端点构建的前端用户界面

（UI），从而使这些端点更易于使用，而且有些实时数据是很难直接通过调用Actuator使用的。

17.1　使用Spring Boot Admin

我曾经被问到很多次，开发一个消费Actuator端点的Web应用并为其提供一个易于查看的UI到底有多么难，这样是否有意义。我的答复是它只是一个REST API，因此所有的事情都是有可能的。不过，当位于德国的软件和咨询公司codecentric AG的优秀工程师已经完成了这项工作时，我们为什么还要为Actuator创建自己的UI呢？

Spring Boot Admin是一个管理类的Web前端应用，使得Actuator的端点更易于被人类所使用。它分为两个主要的组件：Spring Boot Admin服务器和它的客户端。Admin服务器负责收集并展现Actuator数据，而展现的数据则是由一个或多个Spring Boot应用提供的，这些应用就是Spring Boot Admin的客户端，如图17.1所示。

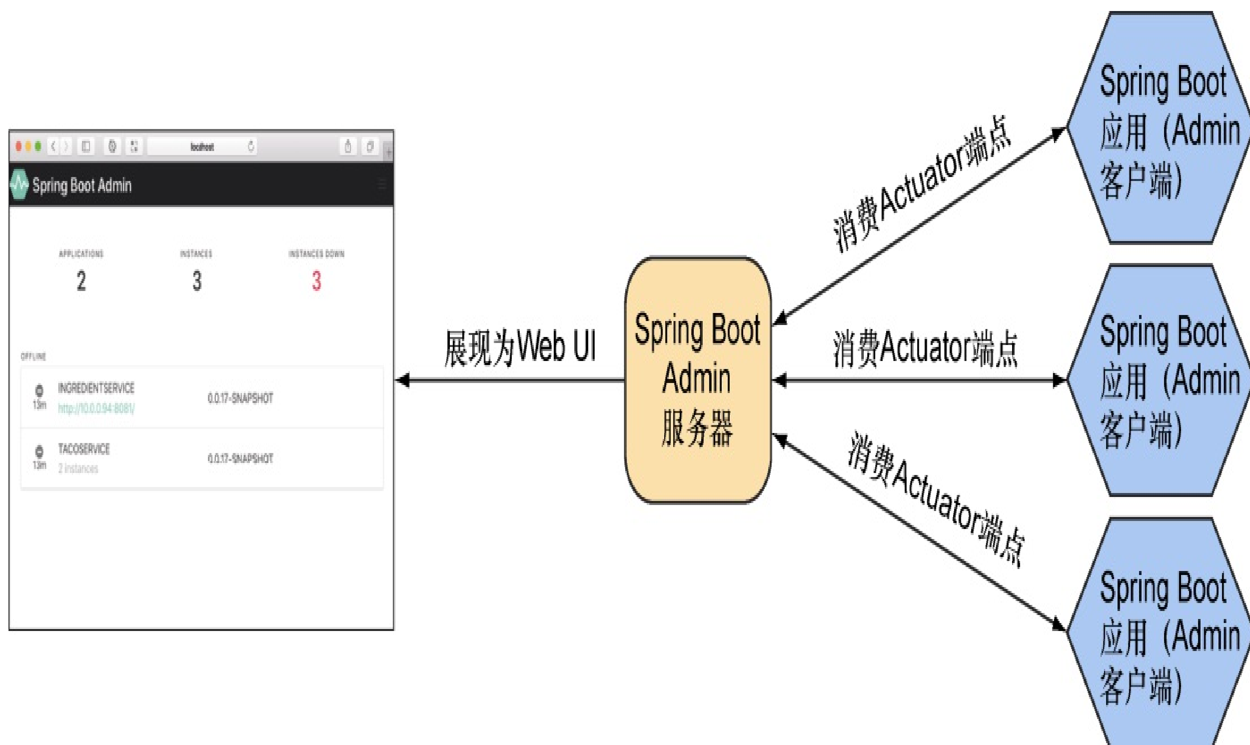


图17.1　Spring Boot Admin的服务器消费来自一个或多个Spring Boot应用的Actuator端点，并将数据展现在一个基于Web的UI中

我们需要将组成Taco Cloud的每个应用（微服务）注册为Spring Boot Admin的客户端。首先，我们需要搭建Spring Boot Admin服务器，以便于接收每个客户端的Actuator信息。

17.1.1　创建Admin服务器

为了启用Admin服务器，我们首先需要创建一个新的Spring Boot应用并将Admin服务器依赖添加到项目的构建文件中。Admin服务器通常会作为一个单独的应用，与其他的应用区分开来。因此，最简单的方式就是使用Spring Boot Initializr创建一个新的Spring Boot项目并选择标签为Spring Boot Admin (Server)的复选框。这样会将如下的依赖添加到

<dependencies>代码块中：

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

现在，我们需要启用Admin服务器，只需要在主配置类上添加@EnableAdminServer注解就可以了，如下所示：

```
package tacos.bootadmin;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import de.codecentric.boot.admin.server.config.EnableAdminServer;

@SpringBootApplication
@EnableAdminServer
public class BootAdminServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(BootAdminServerApplication.class, args);
  }
}
```

最后，因为在开发阶段Admin服务器并不是唯一一个在本地运行的应用，所以我们需要将其设置为监听一个唯一的端口，而且这个端口要易于访问（比如，不能是0）。在这里，我选择9090作为Spring Boot Admin服务器的端口：

```
server:
  port: 9090
```

注意：与其他微服务架构的Spring Boot应用类似，server.port可以在生产环境的profile使用不同的端口，在那时端口

可能会由底层平台来决定。

现在，我们的Admin服务器已经准备就绪。如果此时启动应用并在浏览器中访问http://localhost:9090，那么我们将会看到如图17.2所示的效果。

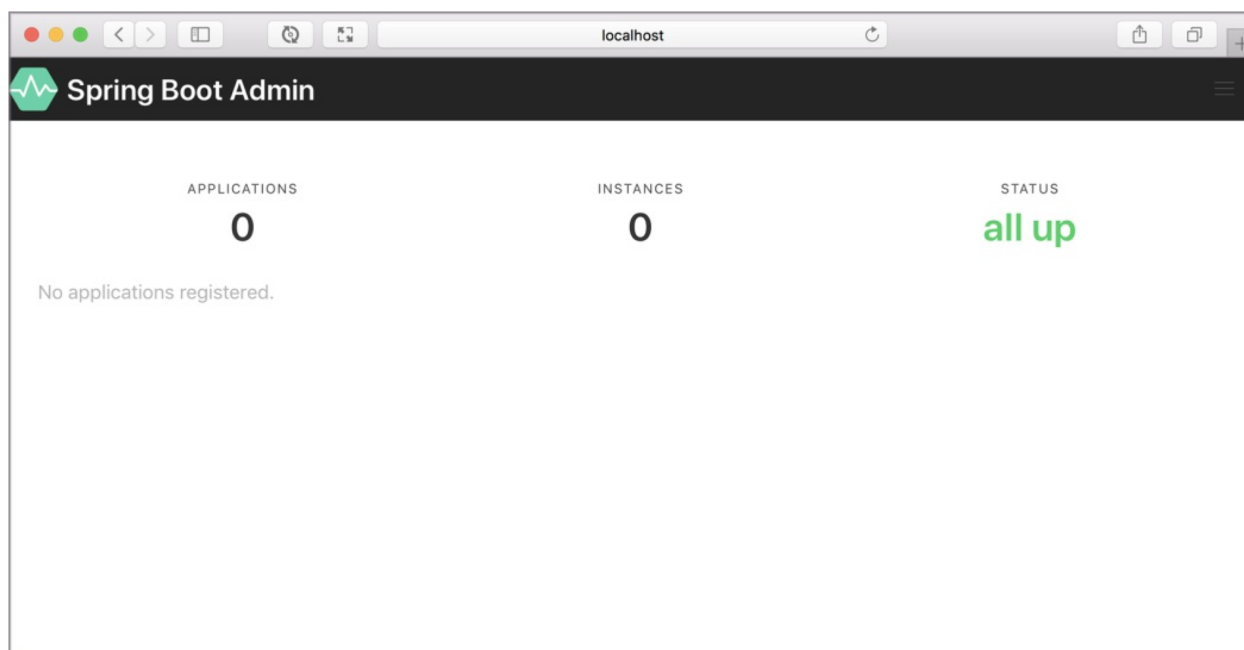


图17.2　没有任何实例在运行

我们可以看到，Spring Boot Admin显示有零个应用的零个实例正在运行。数字下面有“No applications registered.”这样的提示信息，说明此时这些数字没有任何意义。要让Admin服务器真正发挥作用，我们需要为其注册应用。

17.1.2　注册Admin客户端

因为Admin服务器独立于要展现Actuator 数据的其他Spring Boot应用，所以必须让Admin服务器能够以某种方式感知这些应用。Admin服务器注册Spring Boot Admin客户端有两种方式：

- 每个应用显式向Admin服务器注册自身；
- Admin通过Eureka服务注册中心发现服务。

接下来，我们分别看一下这两种方案，首先是如何将单个Spring Boot应用配置为Spring Boot Admin客户端，这样它们就能向Admin服务器注册自身了。

显式配置**Admin**客户端应用

为了让Spring Boot应用注册为Admin服务器的客户端，我们必须将Spring Boot Admin client starter添加到项目的构建文件中。在Initializr中，我们可以选中标签为Spring Boot Admin (Client)的复选框，这样很容易就能将这个依赖添加到构建文件中。对于Maven构建的Spring Boot应用，我们也可以设置如下的依赖：

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

客户端库准备就绪之后，我们需要配置Admin服务器的位置。这样的话，客户端就可以将自身注册进去。为了实现这一点，我们可以将spring.boot.admin.client.url属性设置为Admin服务器的根路径：

```
spring:
```

```
application:
  name: ingredient-service
boot:
  admin:
    client:
      url: http://localhost:9090
```

注意，在这里，我们还设置了spring.application.name属性（在本例中，也就是配料服务）。我们之前已经使用这个属性在Spring Cloud Config Server和Eureka中识别微服务。这里，它的目的很类似：识别Admin服务器中的应用。我们重启应用之后，将会看到它出现在Admin服务器中，如图17.3所示。

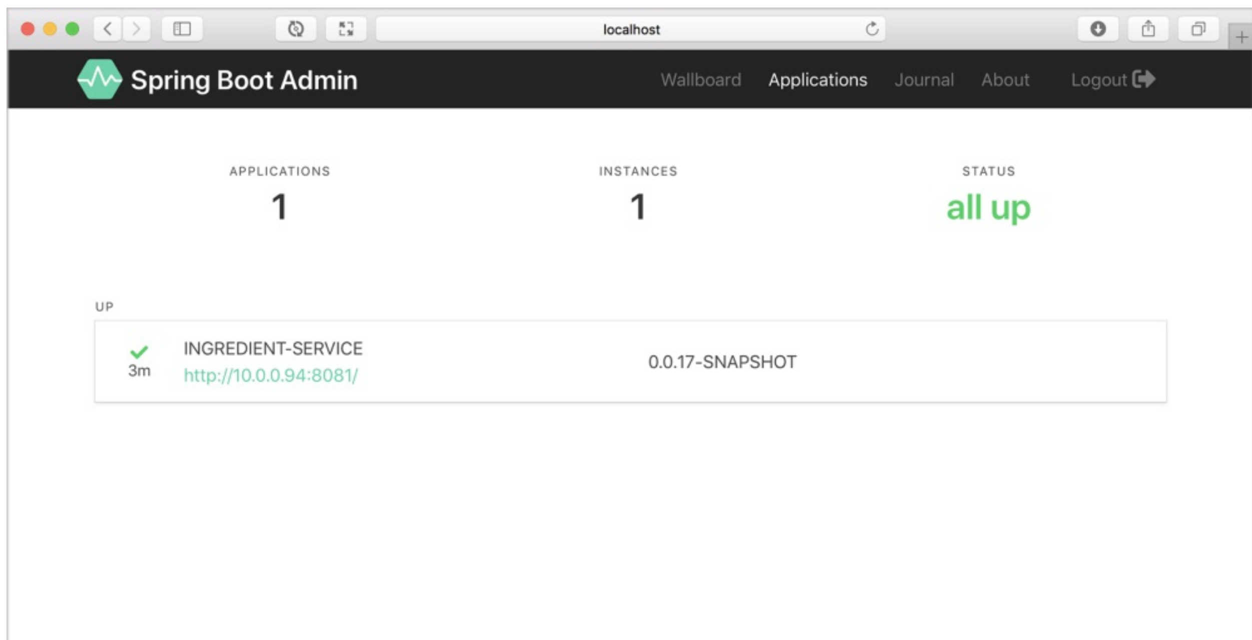


图17.3　Spring Boot Admin UI展现了一个已注册的应用

尽管在图17.3中并没有太多关于配料服务的信息，但是我们可以看到应用的启动时间。如果Spring Boot Maven插件配置了build-info goal（如16.3.1小节所讨论的那样），这里还会显示构建版本。请放

心，在Admin服务器中点击应用之后，我们会看到很多其他运行时的细节。我们将会在17.2节深入了解Admin服务器都提供了哪些功能。

我们需要在所有的应用间重复设置这些在Admin服务器中注册配料服务的配置。一种比较简单的方式是我们只配置spring.application.name属性，Spring Cloud Config Server会将spring.boot.admin.client.url发送给它的所有客户端。如果你已经使用Eureka作为服务注册中心，那么更好的方式是让Admin自己去发现服务。接下来，我们看一下如何将Admin配置为Eureka客户端。

发现**Admin**客户端

如果想让Admin服务器来发现服务，唯一需要做的事情就是添加Spring Cloud Netflix Eureka Client starter到Admin服务器的构建文件中。如下是我们需要的Maven <dependency>：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

注意：我们也可以通过在Spring Initializr中选择Eureka Discovery复选框来添加该依赖。

Admin服务器启用了Eureka客户端功能之后，什么事情都不需要做了。我们可以直接略过上文所述的客户端配置，因为Admin会自动发现

注册在Eureka中的服务并展现它们的Actuator数据。如果Eureka中注册了多个Taco Cloud服务，那么它们将会展现在Admin服务器中（参见图17.4）。

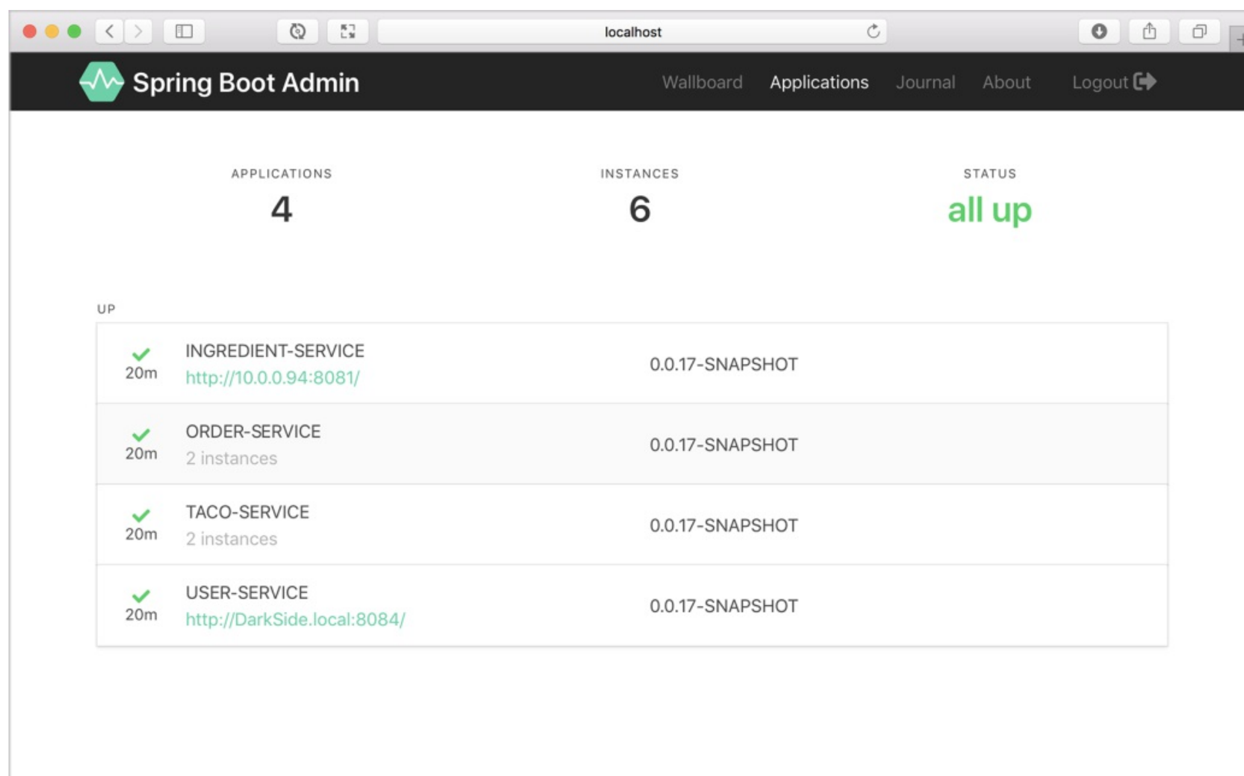


图17.4　Spring Boot Admin UI能够展现在Eureka中发现的所有服务

在图17.4中列出了4个不同的应用，对应了6个服务：订单服务有两个实例，taco服务有两个实例，其他的两个应用各有一个实例。这里显示的所有应用均处于UP状态。如果有应用掉线（比如用户服务），就将会在Admin服务器中单独显示（见图17.5）。

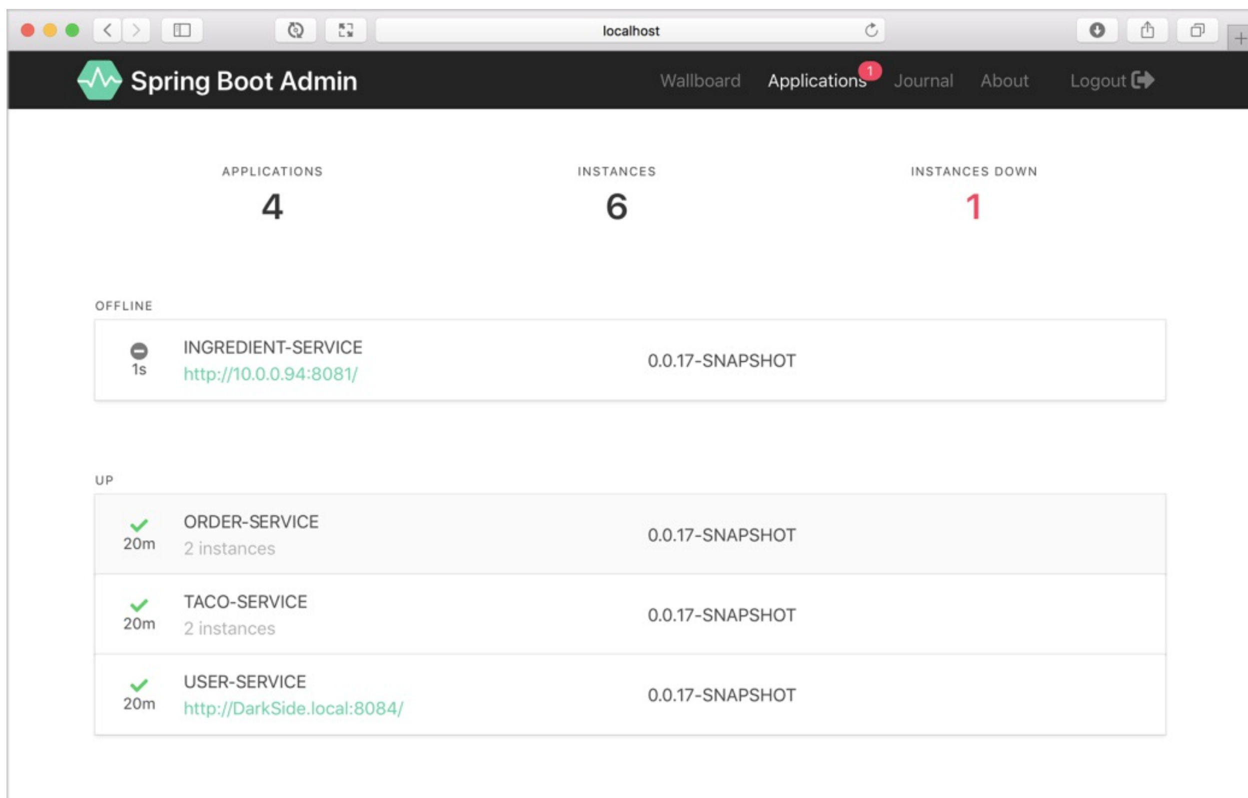


图17.5　Spring Boot Admin UI单独展现掉线的服务，与在线的服务隔离开

作为Eureka的客户端，Admin服务器也会将自身注册为Eureka中的服务。如果要避免这种现象，我们可以将eureka.client.register-with-eureka属性设置为false：

```
eureka:
  client:
    register-with-eureka: false
```

与其他的Eureka客户端类似，如果不是监听默认主机和端口，我们就可以配置Eureka服务器的位置。如下的YAML文件将Eureka位置配置成eureka1.tacocloud.com主机：

```
eureka:
 client:
```

```
    service-url:
      defaultZone: http://eureka1.tacocloud.com:8761/eureka/
```

现在，我们已经将多个Taco Cloud服务注册到了Admin服务器中。接下来，我们看一下Admin服务器都提供了哪些功能。

17.2 探索Admin服务器

在将所有的Spring Boot应用注册为Admin服务器的客户端之后，我们就可以使用Admin服务器得到运行中应用的大量信息，包括：

- 通用的健康信息；
- 通过Micrometer和“/metrics”端点发布的所有指标；
- 环境属性；
- 包和类的日志级别；
- 线程跟踪细节；
- HTTP请求的跟踪情况；
- 审计日志。

实际上，几乎Actuator暴露的所有内容我们都可以通过Admin服务器来查看，只不过它的展现形式更加人性化。它包括了图表和钻取信息的过滤器。Admin服务所展现的信息要比本章中看到的多得多，限于篇幅，我们使用本节剩余的内容着重介绍一些Admin服务器的亮点功能。

17.2.1 查看应用基本的健康状况和信息

正如我们在16.2.1小节所提到的那样，Actuator会通

过“/health”和“/info”端点提供应用的健康状况和基本信息。Admin服务器在Details选项卡下展现了这些信息，如图17.6所示。

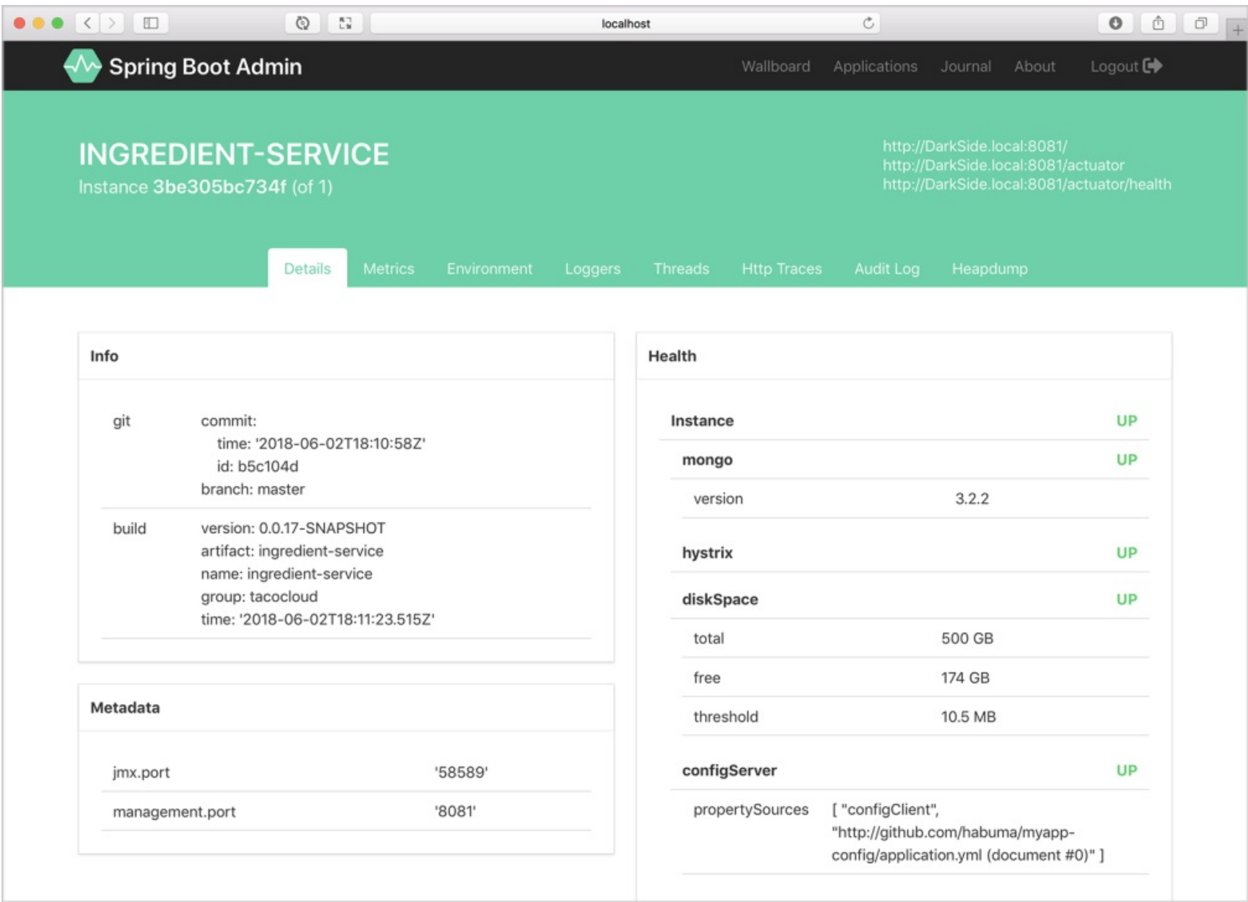


图17.6　Spring Boot Admin UI的Details选项卡展现了应用的健康状况和基本信息

滑过Details选项卡的Health和Info部分，我们会在下方看到一些来自应用JVM的统计信息，包括展现处理器、线程和内存使用的图表，如图17.7所示。

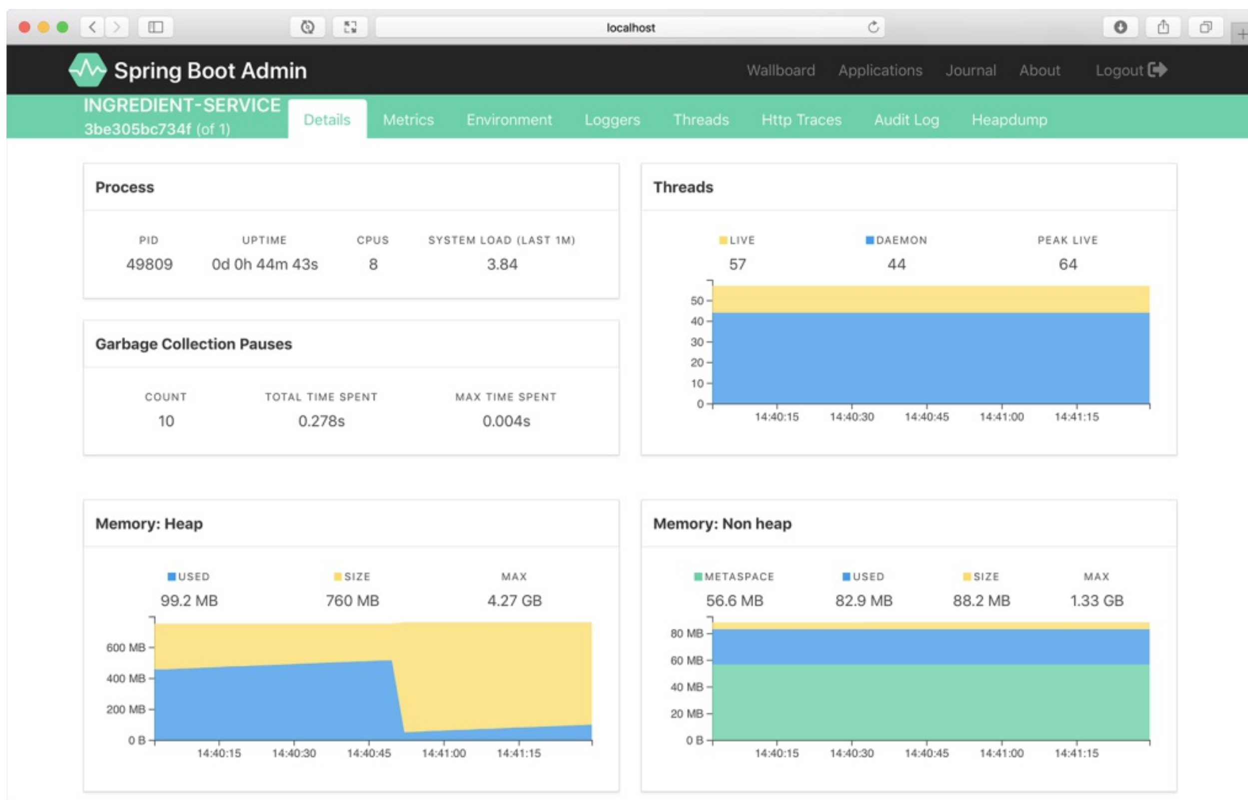


图17.7　在Details选项卡中下方将会看到额外的JVM内部信息（包括处理器、线程和内存统计数据）

图表中所展现的信息再加上Processes、Garbage Collection Pauses下面的指标，可以提供关于应用如何使用JVM资源的有用信息。

17.2.2　观察核心指标

在Actuator的所有端点中，“/metrics”端点所提供的信息可能最不易于人类阅读了。借助Admin服务器Metrics选项卡下的UI界面，我们可以很容易地消费应用所生成的指标数据。

在开始的时候，Metrics选项卡并不会展示任何指标。借助页面顶部

的表单，我们能够设置想要查看的一个或多个指标。

在图 17.8中，我们监视了http.server.requests分类的两个指标：第一个报告展现了发往“/ingredients”端点的HTTP GET请求，并且要求返回状态为200 (OK)；第二个报告展现了所有产生HTTP 404 (NOT FOUND)响应的请求。

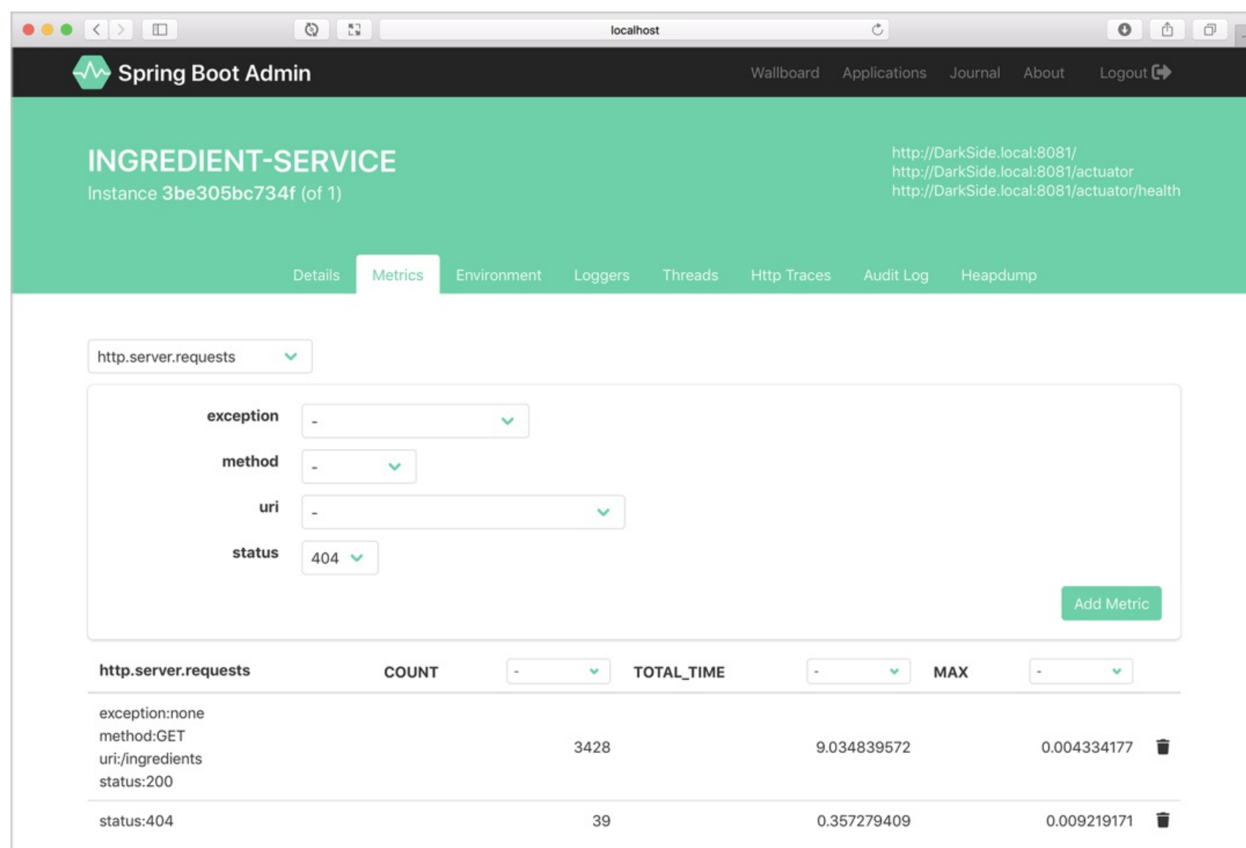


图17.8　在Metrics选项卡下，我们可以监视应用的“/metrics”端点发布的所有指标

关于这些指标，非常棒的一点在于（其实几乎适用于Admin服务器展现的所有内容），这里所展示的是实时数据，会自动更新，无须刷新页面。

17.2.3　探查环境属性

Actuator的“/env”端点能够返回Spring Boot应用所有可用的环境变量，这些环境变量来源于各种属性源。尽管API端点的JSON格式响应并不难读，但是Admin服务器在Environment选项卡下以更美观的形式进行了展现（见图17.9）。

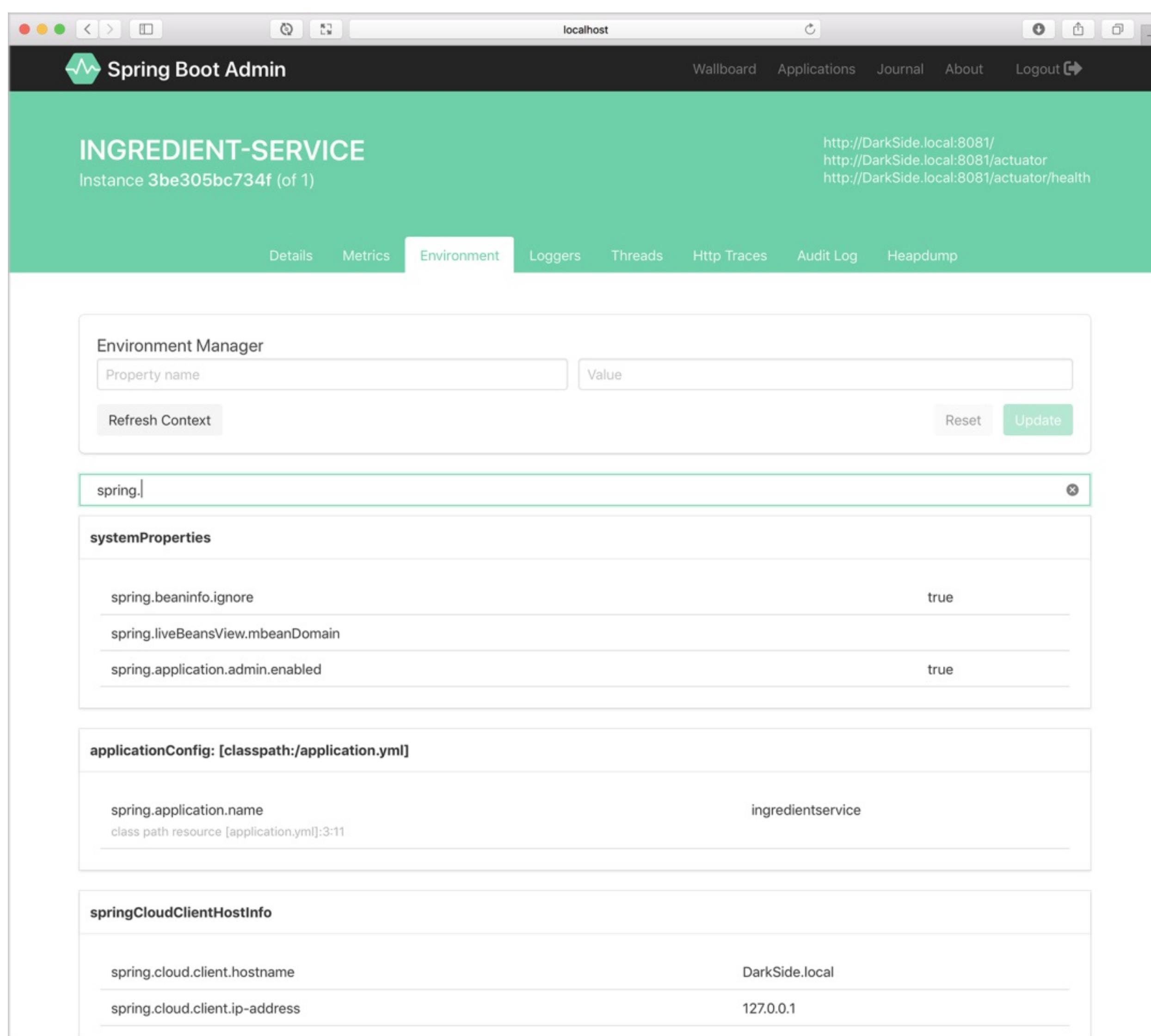


图17.9　Environment选项卡展现了环境属性，并且包含了重写和过滤值的选项

因为这里可能会有数百个属性，所以可以使用属性名或值对可用属性进行过滤。图17.9展现了根据属性名和/或值包含“spring.”进行过滤后的属性列表。通过页面顶部的Environment Manager表单，Admin服务器还允许我们设置或重写环境属性。

17.2.4　查看和设置日志级别

Actuator的“/loggers”端点对于理解或重写运行中应用的日志级别非常有用。Admin服务器的Loggers选项卡基于“/loggers”端点提供了一个非常易于使用的UI页面，进一步简化了应用中的日志管理。图17.10展现了根据org.springframework.boot名称过滤后的loggers。

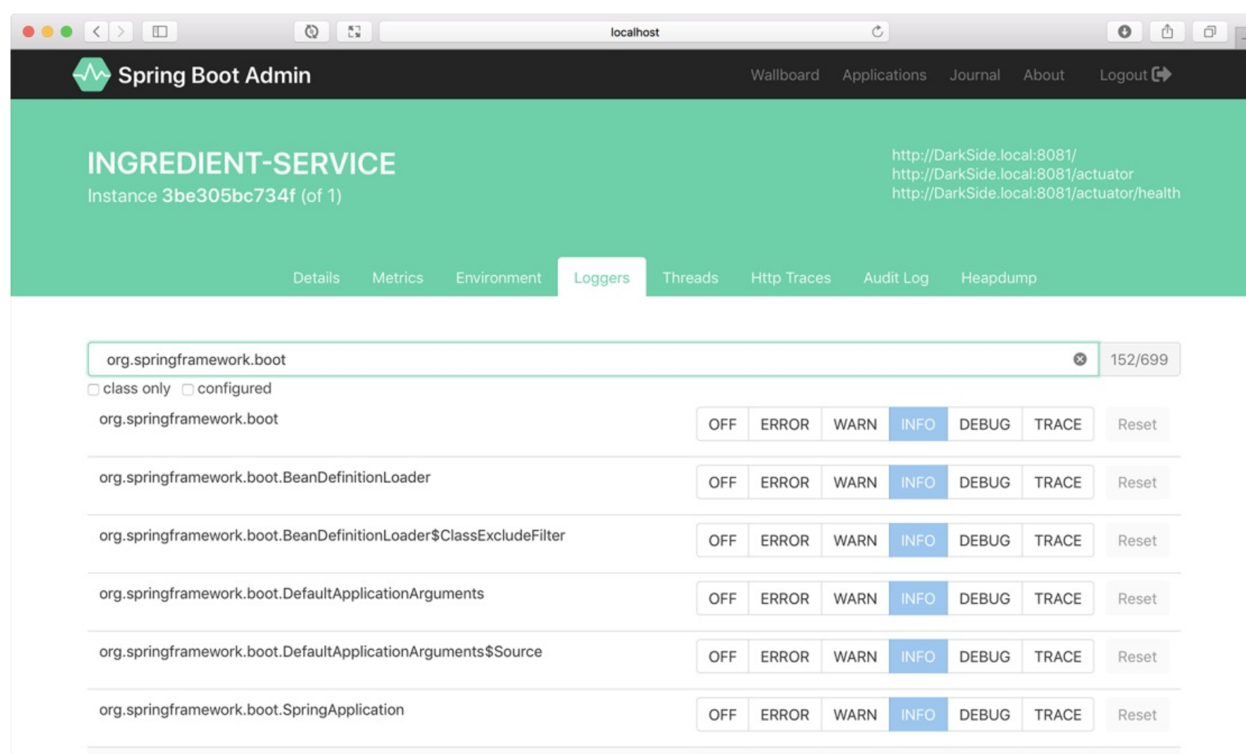


图17.10　Loggers选项卡会展示应用中包和类的日志级别，并且允许我们重写它们的级别

默认情况下，Admin服务器会展现所有包和类的日志级别。它们可以通过名称（仅限于类）和/或显式配置的日志级别来进行过滤，这里不支持对由根logger继承来的级别进行过滤。

17.2.5 监控线程

在应用中，多个线程可以并行运行。尽管“/threaddump”端点（在16.2.3小节进行过描述）提供了应用运行中线程状态的快照，但是Spring Boot Admin UI中的Threads选项卡能够实时查看应用中所有的线程（见图17.11）。

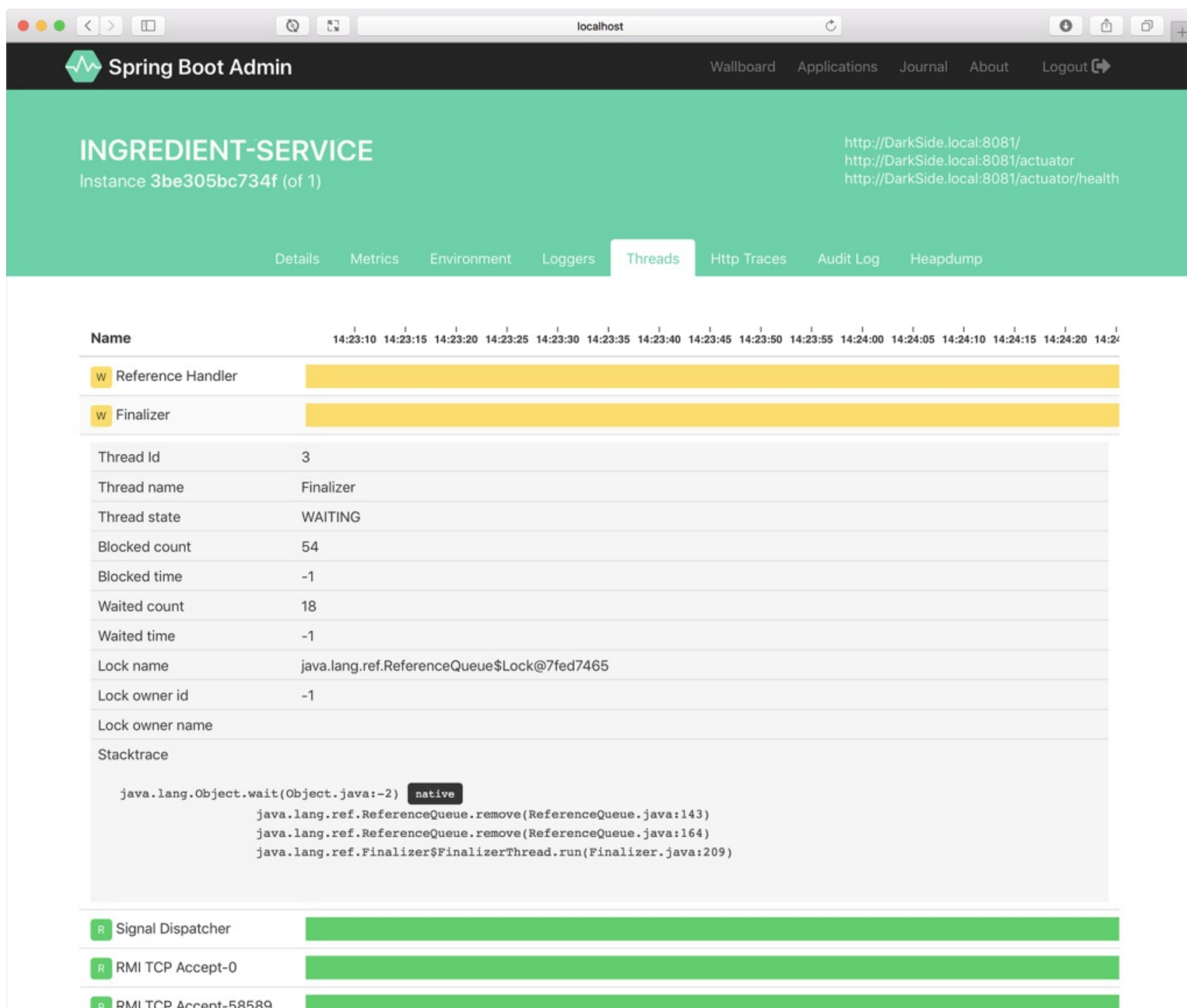


图17.11　我们可以使用Admin UI的Threads选项卡实时监控应用的线程

“/threaddump”端点只是捕获某个时刻的快照，Threads选项卡中的条形图与之不同，它是持续更新的，展示每个线程的状态：线程可运行的话，是绿色的；等待的话，是黄色的；阻塞的话，是红色的。

要查看某个线程的细节信息，可以点击列表中的线程行。这样会显示该线程的历史数据，包括线程当前的堆栈。

17.2.6　跟踪HTTP请求

Spring Boot Admin UI的Http Traces选项卡（见图17.12）展现了Actuator“/httptrace”端点的数据。与“/httptrace”端点返回请求时最近的100个请求不同，Http Traces选项卡列出了完整的HTTP请求历史。而且，在我们打开这个选项卡的时候，数据会一直刷新。如果你离开这个选项卡再回来，那么它初始会显示100条最近的请求，但是会从当前时刻开始进行跟踪。

我们可以看到，Http Traces选项卡包含了一个随时间变化的HTTP流量的堆积图（stacked graph）。这个图使用不同的颜色来表示成功和失败的请求：绿色代表成功，黄色代表客户端错误（例如404级别的HTTP响应），红色代表服务器错误（如500级别的HTTP响应）。如果将鼠标指针移动到图上，就会弹出一个悬停框（如图17.12最右侧所示），显示给定时间分解的请求计数。

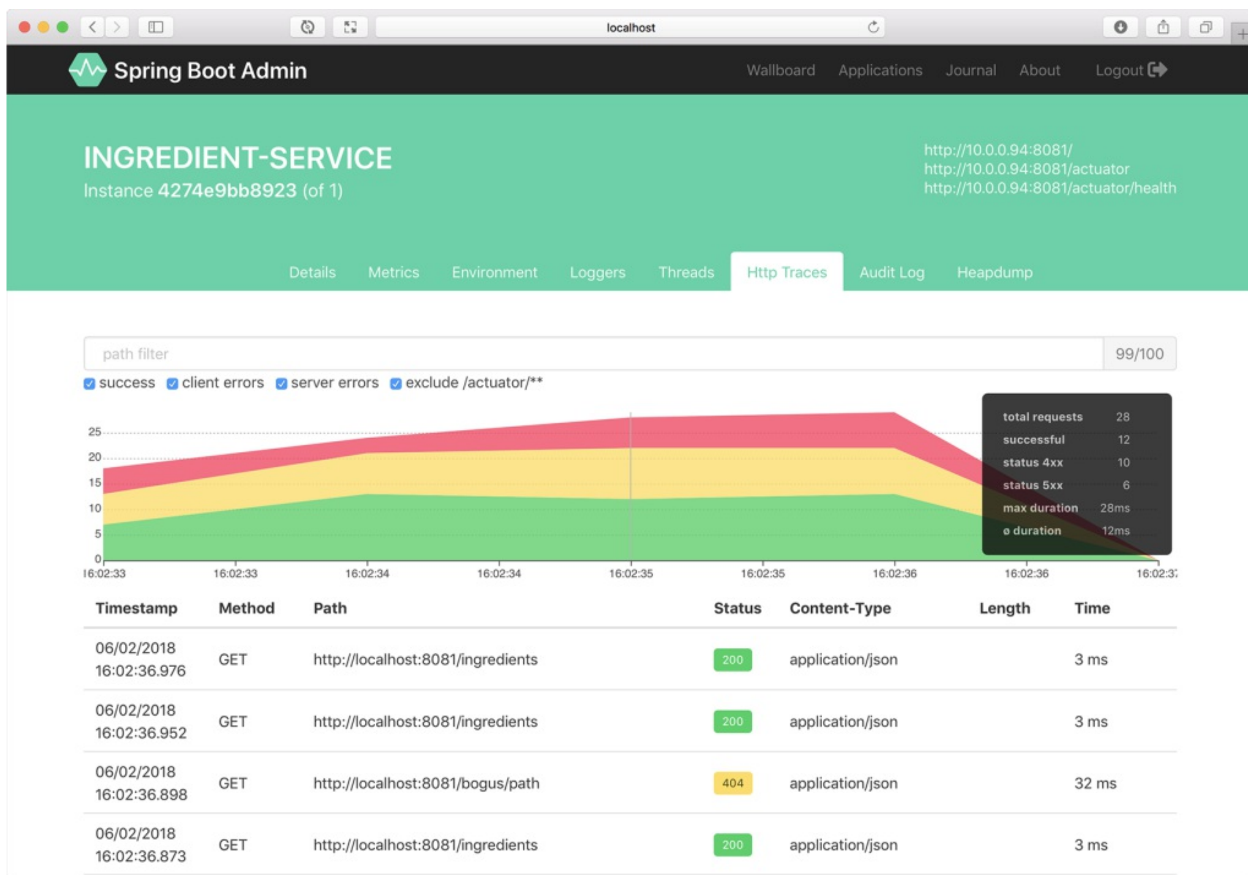


图17.12　Http Traces选项卡会跟踪该应用最近的HTTP流量，包括产生错误的请求信息

在堆积图的下方展现了跟踪历史，应用接收到的每个请求都会对应一行。点击其中一行，就会展开显示该请求的额外数据，包括请求和响应的头信息（见图17.13）。

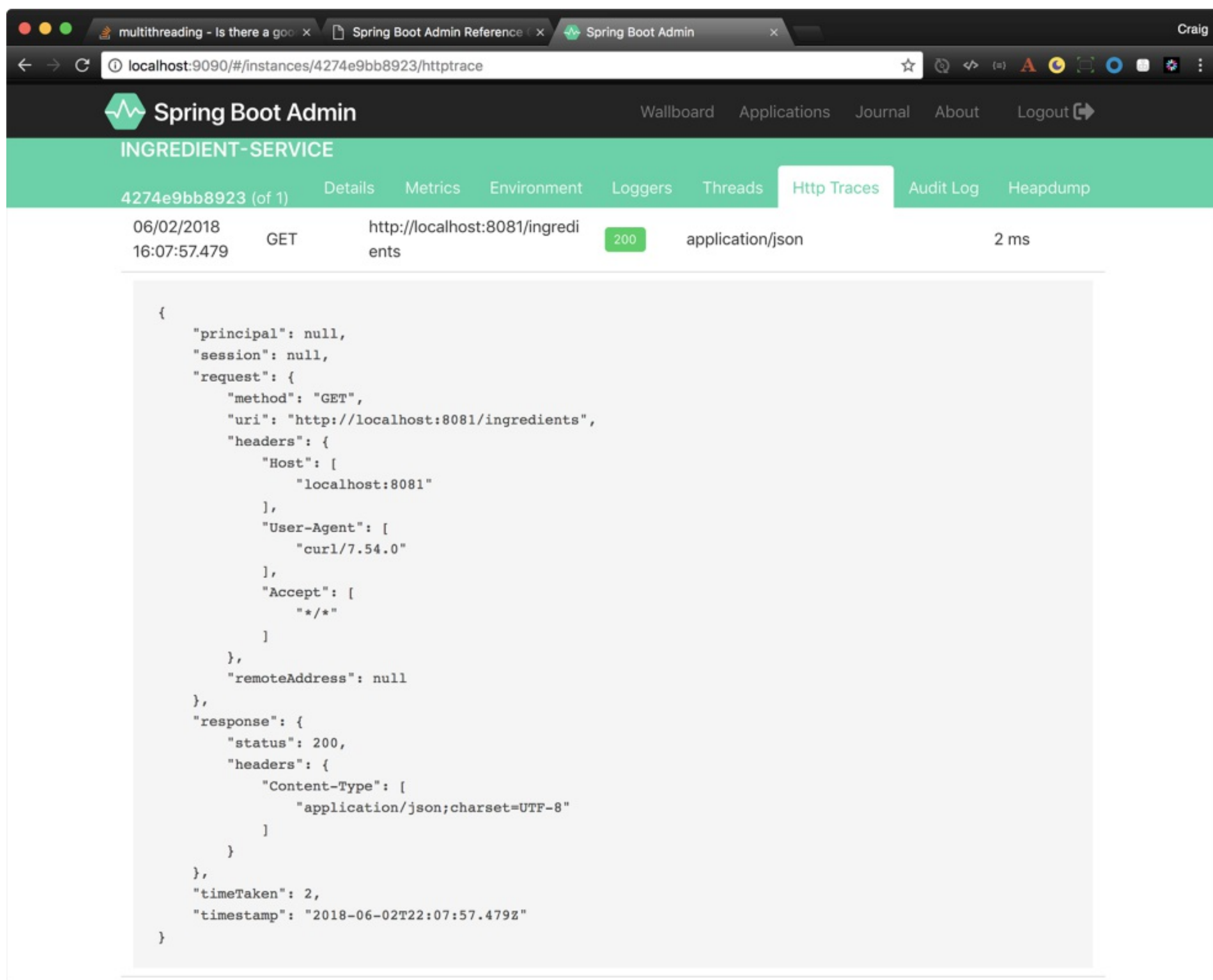


图17.13　点击Http Traces选项卡中的请求条目，将会展现该请求额外的详情

17.3　保护Admin服务器

正如我们在第16章所讨论的那样，Actuator端点对外暴露的信息并不能随便消费。它们包含的信息暴露了应用的详情，这些信息只有应用程序的管理员才能查看。另外，还有一些端点允许对应用进行变更，它们就更不应该对所有人开放了。

正如安全性对于Actuator来说非常重要一样，它对Admin服务器同

样重要。除此之外，如果Actuator端点需要认证，那么Admin需要知道凭证信息才能访问这些端点。接下来，我们看一下如何为Admin服务器添加一些安全性。首先，从认证开始。

17.3.1 为Admin服务器启用登录功能

默认情况下，Admin服务器是不安全的，所以为其添加安全性功能是一种好的做法。因为Admin服务器就是一个Spring Boot应用，所以我们可以使用Spring Security来保护它。这一点与其他的Spring Boot应用完全类似。就像使用Spring Security保护其他的应用一样，我们可以自由选择最适合需求的安全模式。

按照最小的要求，我们需要添加Spring Boot security starter到Admin服务器的构建文件中，既可以在Initializr中选中Security复选框，也可以添加如下的<dependency>到项目的pom.xml文件中：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

然后，为了避免观察Admin服务器的日志才能获取随机生成的密码，我们可以将简单的管理员用户名和密码配置在application.yml中：

```
spring:
  security:
    user:
      name: admin
      password: 53cr3t
```

现在，当在浏览器中加载Admin服务器的时候，我们会看到Spring Security默认的登录表单，提示我们输入用户名和密码。按照这里的配置片段，输入“admin”和“53cr3t”就可以登录了。当然，这是一个极其基本的安全配置。我推荐你参考第4章了解配置Spring Security的各种方式，为Admin服务器提供更丰富的安全模式。

17.3.2　为Actuator启用认证

在16.4节中，我们讨论了如何使用HTTP Basic认证保护Actuator端点。按照这种方式，我们会将不知道Actuator端点用户名和密码的用户拒之门外。也就意味着Admin服务器不能消费Actuator端点了，除非提供用户名和密码。但是，Admin如何得到凭证信息呢？

Admin服务器的客户端应用可以通过直接向Admin服务器注册自身或被Eureka发现的方式提供凭证信息给Admin服务器。如果应用是直接向Admin服务器注册自身，那么可以在注册时发送凭证信息。我们需要配置几个属性启用该功能。

spring.boot.admin.client.instance.metadata.user.name和spring.boot.admin.client.instance. metadata.user.password属性指定了Admin服务器访问应用的Actuator端点时可以使用的凭证信息。application.yml中如下的代码片段展示了如何设置这些属性：

```
spring:
  boot:
    admin:
      client:
```

```
      url: http://localhost:9090
      instance:
        metadata:
          user.name: ${spring.security.user.name}
          user.password: ${spring.security.user.password}
```

用户名和密码必须要设置在所有向Admin服务器注册的应用中。这里给定的值必须要匹配Actuator端点HTTP Basic认证头信息所需的用户名和密码。在本例中，它们设置成了admin和password，也就是访问Actuator端点所配置的凭证信息。

如果应用是由Admin服务器通过Eureka发现的，那么我们需要设置eureka.instance.metadata-map.user.name和eureka.instance.metadata-map.user.password：

```
eureka:
  instance:
    metadata-map:
      user.name: admin
      user.password: password
```

当应用使用Eureka注册的时候，凭证信息将会包含到Eureka注册记录的元数据中。当Admin服务器发现应用时，会和应用的其他详情一起从Eureka获取它的凭证信息。

17.4 小结

- Spring Boot Admin服务器能够消费一个或多个Spring Boot应用的Actuator端点，并在一个用户友好的Web应用中展现数据。
- Spring Boot可以向Admin服务器注册自身，也可以通过Eureka被Admin服务器自动发现。

- 与捕获应用状态快照的Actuator端点不同，Admin服务器可以展现应用内部运行状况的实时视图。
- 借助Admin服务器能够很容易地过滤Actuator结果，在有些场景下，还可以以可视化图表的形式展现数据。
- 因为Admin服务器就是一个Spring Boot应用，所以可以使用任意可用的Spring Security方式来保护它。

第18章　使用JMX监控Spring

本章内容：

- 使用Actuator端点的MBean
- 将Spring bean暴露为MBean
- 发布通知

JMX（Java Management Extensions，Java管理扩展）作为监视和管理Java应用程序的标准方法已经存在超过了15年。通过暴露名为MBean（托管bean）的托管组件，外部的JMX客户端可以通过调用MBean中的操作、探查属性和监视事件来管理应用程序。

在Spring Boot应用中，JMX会自动启用。这样的话，Actuator的所有端点均会暴露为MBean。另外，它还会搭建一个很便利的环境，能够很容易地将Spring应用上下文中的bean暴露为MBean。作为探索Spring和JMX功能的开始，我们首先看一下Actuator端点是如何暴露为MBean的。

18.1　使用Actuator MBean

我们可以回头看一下表 16.1，除了“/heapdump”之外，这里列出的所有端点均暴露成了MBean。我们可以使用任意的JMX客户端连接Actuator端点MBean。借助Java开发工具集中的JConsole，我们可以看到Actuator MBean列到了org.springframework.boot域下，如图18.1所示。

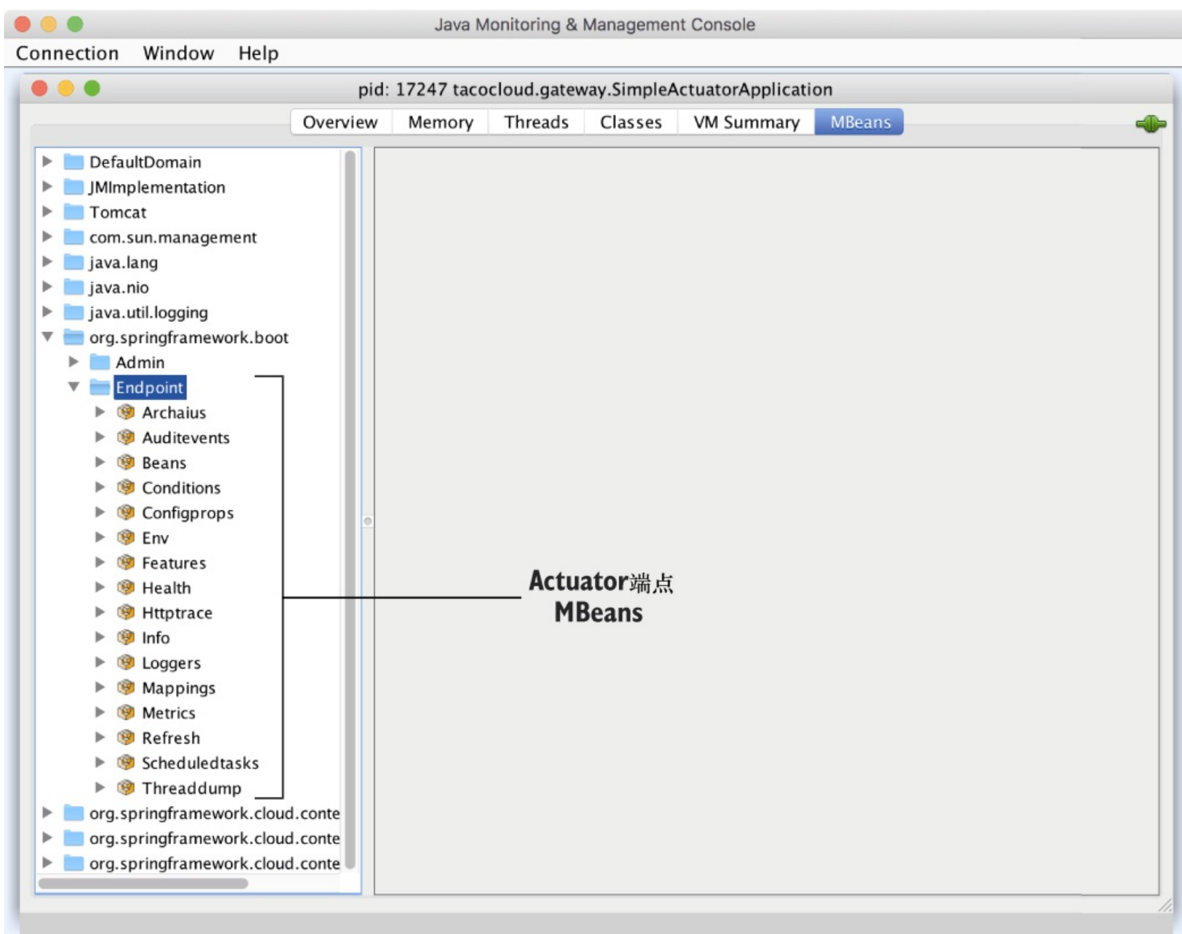


图18.1　Actuator端点会自动暴露为JMX MBean

Actuator MBean端点非常好的一点在于它们默认就是对外暴露的。我们没有必要明确声明要包含哪些MBean端点，但是对于HTTP端点，

我们是需要这样做的。我们可以通过设置management.endpoints.jmx.exposure.include和management.endpoints.jmx.exposure.exclude属性来缩小可选的范围。例如，我们想要限制Actuator端点只暴露“/health”“/info”“/bean”和“/conditions”端点，那么可以按照如下的方式设置management.endpoints.jmx.exposure.include：

```
management:
  endpoints:
    jmx:
      exposure:
        include: health,info,bean,conditions
```

我们只想排除其中的几个端点的话，可以按照如下的方式设置management.endpoints. jmx.exposure.exclude属性：

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: env,metrics
```

在这里，我们使用management.endpoints.jmx.exposure.exclude排除了“/env”和“/metrics”端点。所有其他的Actuator端点依然会暴露为MBean。

要在JConsole中调用一个或多个Actuator MBean所托管的操作，可以在左侧树中展开MBean端点，然后在Operations下选择所需的操作。

例如，你想要探查tacos.ingredients包的日志级别，那么可以展开Loggers MBean并点击名为loggerLevels的操作，如图18.2所示。在右上方的表单中，在name文本域中输入包名（tacos.ingredients），然后点击

loggerLevels按钮。

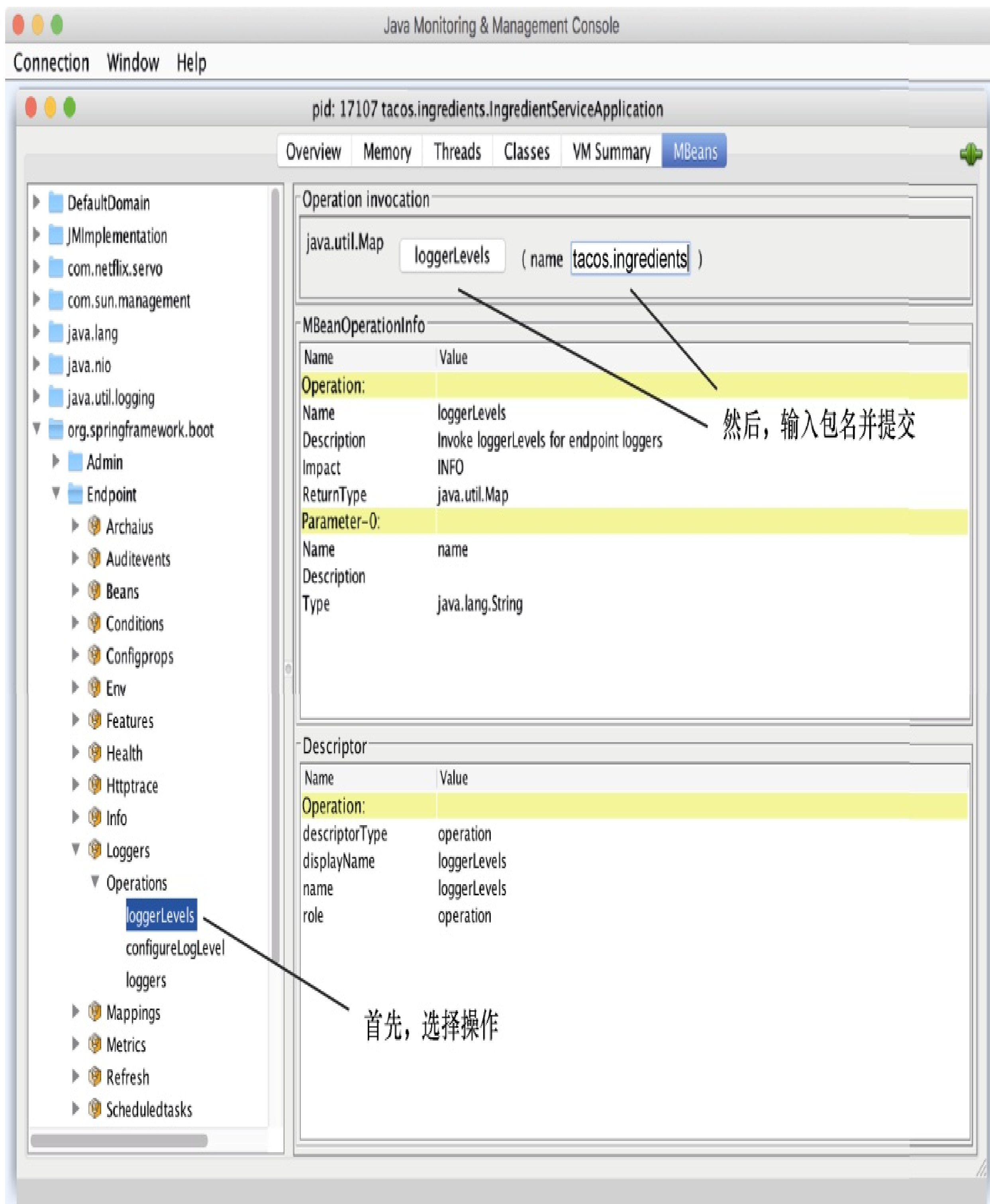


图18.2　使用JConsole展现Spring Boot应用的日志级别

在点击loggerLevels按钮之后，将会弹出一个对话框，展现来自“/loggers”端点MBean的响应，大致如图18.3所示。

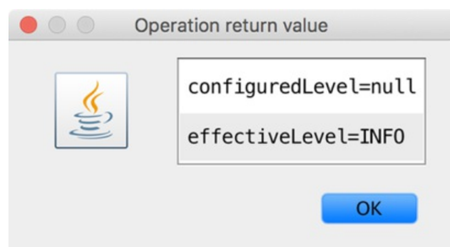


图18.3　在JConsole中，“/loggers”端点MBean所展现的日志级别

尽管JConsole UI的使用方式有些笨拙，但是你应该可以掌握它的技巧并使用相同的方式来探索其他的Actuator端点。

如果你不喜欢JConsole，也没有问题，有很多其他的JMX客户端可供选择。

18.2　创建自己的MBean

借助Spring，可以很容易地将任意bean导出为JMX MBean。我们唯一需要做的就是在bean类上添加@ManagedResource注解，然后在方法或属性上添加@ManagedOperation或@ManagedAttribute。Spring会负责剩余的事情。

例如，我们想要提供一个MBean，用来跟踪通过Taco Cloud创建了多少个taco订单，那么我们可以定义一个服务bean，在这个服务中保持已创建taco的数量。程序清单18.1展现了该服务。

程序清单18.1　统计已创建taco订单数量的MBean

```
package tacos.tacos;
import java.util.concurrent.atomic.AtomicLong;
import
     org.springframework.data.rest.core.event.AbstractRepositoryEventListe
ner;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Service;

@Service
@ManagedResource
public class TacoCounter
       extends AbstractRepositoryEventListener<Taco> {

  private AtomicLong counter;

  public TacoCounter(TacoRepository tacoRepo) {
    long initialCount = tacoRepo.count();
    this.counter = new AtomicLong(initialCount);
  }

  @Override
  protected void onAfterCreate(Taco entity) {
    counter.incrementAndGet();
  }

  @ManagedAttribute
  public long getTacoCount() {
    return counter.get();
  }

  @ManagedOperation
  public long increment(long delta) {
    return counter.addAndGet(delta);
  }
}
```

TacoCounter类使用了@Service注解，所以它将会被组件扫描功能所发现并且会注册一个实例作为bean存放到Spring应用上下文中。它还使用了@ManagedResource注解，表明这个bean是一个MBean。作为MBean，它暴露了一个属性和一个操作。getTacoCount()方法使用了

@ManagedAttribute注解，将会暴露为一个MBean属性；而increment()方法使用了@ManagedOperation注解，将会暴露为MBean操作。

图18.4展现了TacoCounter MBean在JConsole中是什么样子。

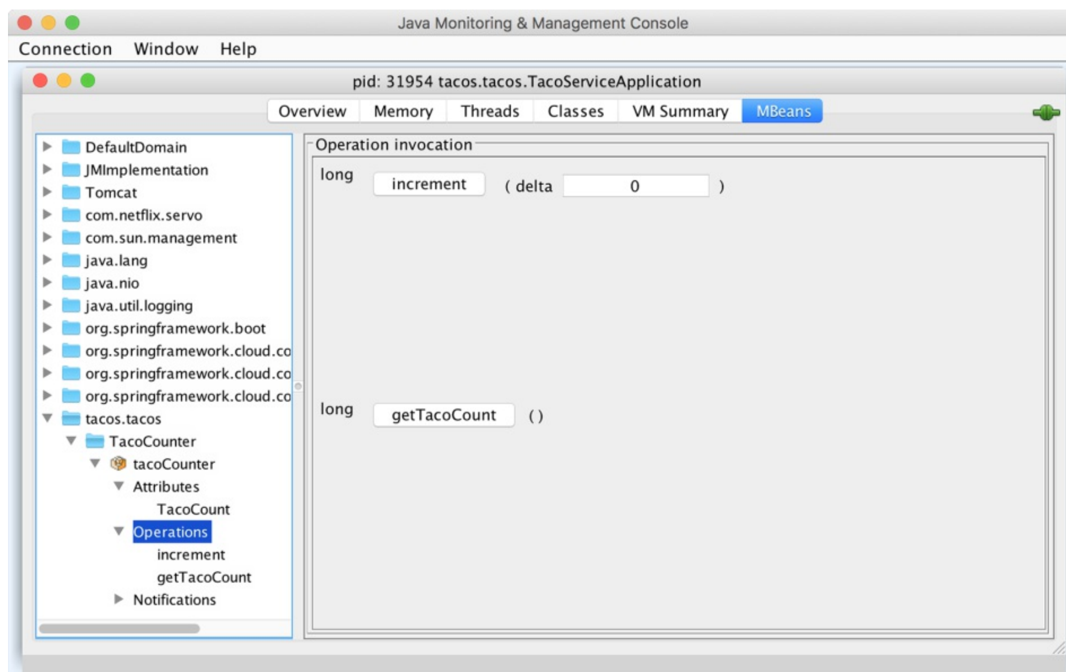


图18.4　在JConsole中看到的TacoCounter的操作和属性

TacoCounter还有一个技巧，不过这与JMX并没有什么关系。这个类扩展了AbstractRepositoryEventListener，每当通过TacoRepository保存Taco的时候，它都会得到通知。在本例中，在创建和保存新Taco对象的时候，onAfterCreate()方法将会被调用，我们在这里让计数器增加 1。AbstractRepositoryEventListener还提供了多个方法来处理对象创建、保存和删除前后的事件。

使用MBean的操作和属性在很大程度上是一个拉取操作。换句话说，就算MBean属性的值发生了变化，除非通过JMX客户端查看该属

性，否则我们也不会知道。接下来，我们换一个话题，看一下如何将MBean的通知推送至JMX客户端。

18.3　发送通知

借助Spring的NotificationPublisher，MBeans可以推送通知到感兴趣的JMX客户端。NotificationPublisher有一个sendNotification()方法，当得到一个Notification对象时，它会发送通知给任意订阅该MBean的JMX客户端。

要让某个MBean发送通知，它必须要实现NotificationPublisherAware接口，该接口要求实现一个setNotificationPublisher()方法。例如，我们希望每创建100个taco就发送一个通知。我们可以修改TacoCounter类，让它实现NotificationPublisherAware，并使用注入的NotificationPublisher每创建100个taco时就发送通知。程序清单18.2展现了启用通知功能TacoCounter所需要的变更。

程序清单18.2　每创建100个taco就发送通知

```
@Service
@ManagedResource
public class TacoCounter
       extends AbstractRepositoryEventListener<Taco>
       implements NotificationPublisherAware {

  private AtomicLong counter;
  private NotificationPublisher np;

...
```

```
  @Override
  public void setNotificationPublisher(NotificationPublisher np) {
    this.np = np;
  }

...

  @ManagedOperation
  public long increment(long delta) {
    long before = counter.get();
    long after = counter.addAndGet(delta);
    if ((after / 100) > (before / 100)) {
      Notification notification = new Notification(
          "taco.count", this,
          before, after + "th taco created!");
      np.sendNotification(notification);
    }

    return after;
  }
}
```

在JMX客户端中，我们需要订阅TacoCounter MBean来接收通知。每创建100个taco，客户端就会收到通知。图18.5展现了通知在JConsole中的样子。

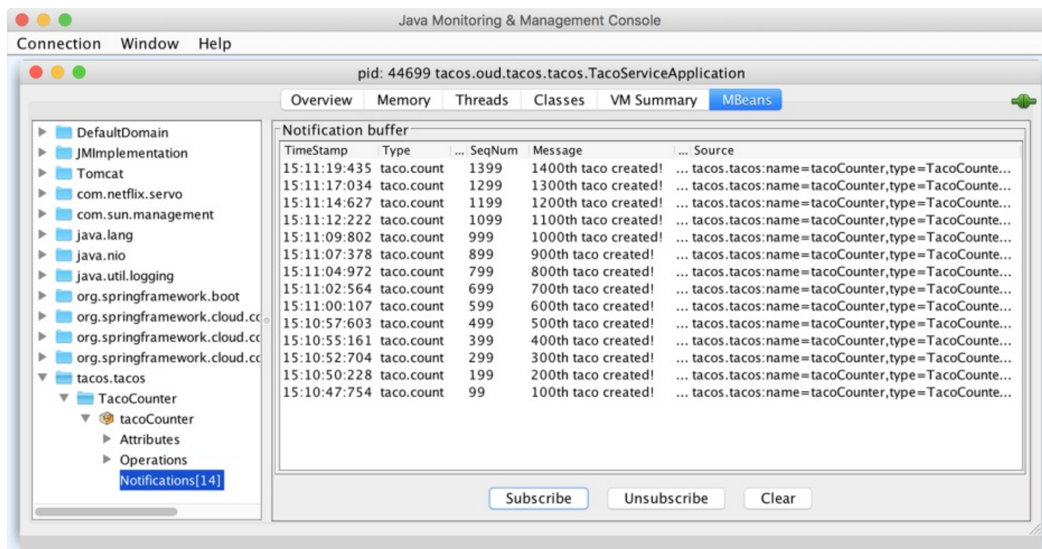


图18.5　JConsole订阅了TacoCounter MBean，每创建100个taco就会收到通知

通知是应用程序主动向监视客户端发送数据和告警的好办法。这样做的话，就不需要客户端轮询托管属性或调用托管操作了。

18.4 小结

- 大多数Actuator端点都可以作为MBean使用，可以被JMX客户端消费。
- Spring会自动启用JMX，用来监控Spring应用上下文中的bean。
- Spring bean可以通过添加@ManagedResource注解导出为MBean。通过为bean类添加@ManagedOperation和@ManagedAttribute注解，它的方法和属性可以导出为托管的操作和属性。
- Spring bean可以使用NotificationPublisher发送通知给JMX客户端。

第19章　部署Spring

本章内容：

- 将Spring应用构建为WAR或JAR文件
- 推送Spring应用至Cloud Foundry
- 使用Docker容器化Spring应用

想一下你最喜欢的动作片。现在我们想象一下，你要去电影院看那部电影，在高速追逐、爆炸和战斗中体验一场激动人心的视听之旅，但是电影最终却在好人打倒坏人之前戛然而止。电影院的灯一亮，所有人都被带出影院，我们没有看到电影里的冲突是如何解决的。虽然开头很精彩，但是重要的是影片的高潮部分。没有它，那就是为了行动而行动。

现在想象一下我们开发了应用程序，并在解决业务问题方面投入了大量的精力和创造力，但是从来没有将应用程序部署给其他人使用和享受。当然，我们编写的大多数应用程序都不涉及汽车追逐或爆炸（至少

我不希望如此），但是在开发过程中会有一定的忙乱。并不是我们所写的每一行代码都是为生产而写的，但是，如果没有任何代码被部署的话，将是极端令人失望的。

到目前为止，我们一直在关注Spring Boot所提供的帮助应用开发的特性。在这个过程中，已经有了一些令人兴奋的进展。如果不越过终点线，也就是部署应用程序，那么这一切都是徒劳的。

在本章中，我们将会在使用Spring Boot开发应用的基础上再进一步，看一下如何部署这些应用。尽管对于部署过基于Java应用的人来说，这些事情是显而易见的，但是Spring Boot以及相关的Spring项目有一些独特之处，它们使得Spring Boot应用的部署与众不同。

实际上，与大多数以WAR文件部署的Java Web应用不同，Spring Boot提供了多种部署方案。在开始学习如何部署Spring Boot应用之前，我们看一下所有的可选方案并选择最适合需求的几种。

19.1　权衡各种部署方案

我们可以以多种方式构建和运行Spring Boot应用。在附录中将介绍其中的一部分，包括：

- 使用Spring Tool Suite或IntelliJ IDEA在IDE中运行应用；
- 在命令行中通过Maven spring-boot:run goal或Gradle bootRun任务运行应用；
- 使用Maven或Gradle生成一个可执行的JAR文件，既可以在命令行

运行，也可以部署到云中；

- 使用Maven或Gradle生成一个WAR文件，以部署到传统的Java应用服务器中。

这些可选方案都非常适合在开发阶段运行应用。但是，如果我们想要将应用部署到生产环境或者其他非开发环境，又该怎么办呢？

通过IDE或者Maven、Gradle运行应用并不适用于生产环境，可执行的JAR文件或者传统的WAR文件才是将应用部署到生产环境的可行方案。既然可以部署为WAR文件或JAR文件，那我们该选择哪种呢？通常，这种选择取决于要将应用部署到传统的Java应用服务器中还是部署到云中。

- 部署到Java应用服务器中：如果必须要将应用部署到Tomcat、WebSphere、WebLogic或其他传统的Java应用服务器中，其实我们别无选择，只能将应用构建为WAR文件。
- 部署到云中：如果你计划将应用部署到云中，不管是Cloud Foundry、Amazon Web Services（AWS）、Azure、Google Cloud Platform还是其他云平台，那么可执行的JAR文件是最佳选择。即便云平台支持WAR部署，JAR文件格式也要比WAR格式简单得多，WAR文件是专门针对应用服务器部署设计的。

在本章中，我们将会关注3种部署场景。

- 将Spring Boot应用以WAR文件的形式部署到Java应用服务器中，比如Tomcat。
- 将Spring Boot应用作为可执行的JAR文件，推送到Cloud Foundry中。

- 将Spring Boot应用打包到Docker容器中，将其部署到任何支持Docker形式的平台中。

首先，我们看一下如何将配料服务应用构建为一个WAR文件，这样它就可以部署到像Tomcat这样的应用服务器中了。

19.2 构建和部署WAR文件

在本书中，我们编写Taco Cloud应用所需的服务时，都是在IDE中运行，或者通过命令行以可执行文件的形式运行。不管是使用哪种方式，都会有一个嵌入式的Tomcat服务器（在Spring WebFlux应用中会是Netty）来为应用的请求提供服务。

在很大程度上，借助Spring Boot的自动配置，我们不需要创建web.xml文件或Servlet initializer类来声明Spring的DispatcherServlet，以实现Spring MVC相关的功能。如果要将应用程序部署到Java应用服务器中，就需要构建一个WAR文件。而且，为了让应用服务器知道如何运行应用程序，我们还需要在WAR文件中包含一个servlet initializer，以扮演web.xml文件的角色并声明DispatcherServlet。

实际上，要将Spring Boot应用构建为WAR文件并不困难。在使用Initializr创建应用的时候，如果选择了WAR方案，其实并没有额外要做的事情了。

Initializr会确保所生成的项目包含servlet initializer类，并且构建文件调整为生成WAR文件。如果你在Initializr中选择了构建为JAR文件（或

者只是想知道它们之间的差异是什么），那么可以继续向下阅读。

首先，我们需要有一种配置Spring DispatcherServlet的方式。虽然可以通过web.xml文件来实现，但是Spring Boot的SpringBootServletInitializer使这个过程变得更加简单了。SpringBootServletInitializer是一个能够感知Spring Boot环境的特殊SpringWebApplicationInitializer实现。除了配置Spring的DispatcherServlet之外，SpringBootServletInitializer还会查找Spring应用上下文中所有Filter、Servlet或ServletContextInitializer类型的bean，并将它们绑定到servlet容器中。

要使用SpringBootServletInitializer，我们需要创建一个子类并重写configure()方法来指明Spring配置类。程序清单19.1展现了IngredientServiceServletInitializer，它是SpringBootServletInitializer的子类，我们将会使用它来实现配料服务应用。

程序清单19.1　通过Java启用Spring Web应用

```
package tacos.ingredients;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class IngredientServiceServletInitializer
       extends SpringBootServletInitializer {
  @Override
  protected SpringApplicationBuilder configure(
                                SpringApplicationBuilder builder) {
    return builder.sources(IngredientServiceApplication.class);
  }
}
```

我们可以看到，configure()方法以参数形式得到了一个SpringApplicationBuilder对象，并且将其作为结果返回。在中间的代码中，它调用sources()方法来注册Spring配置类。在本例中，它注册了IngredientServiceApplication类，这个类同时作为（可执行JAR的）引导类和Spring配置类。

虽然配料服务应用还有其他的Spring配置类，但是我们没有必要将它们全部注册到sources()方法中。IngredientServiceApplication类使用了@SpringBootApplication，说明将会启用组件扫描。组件扫描功能会发现其他的配置类并将它们添加进来。

在大多数情况下，SpringBootServletInitializer的子类都是样板式的。它引用了应用的主配置类。除此之外，在构建WAR时，每个应用都是相同的。我们几乎没有必要去修改它。

现在，我们已经编写完servlet initializer类。接下来，必须要对项目的构建文件做一些修改。如果使用Maven进行构建，那么所需的变更非常简单，只需要确保pom.xml中的<packaging>元素设置成war即可：

```
<packaging>war</packaging>
```

Gradle构建所需的变更也很简单直接，我们需要在build.gradle文件中应用war插件：

```
apply plugin: 'war'
```

现在，我们就可以构建应用了。如果使用Maven，那么我们可以借

助Initializr所使用的Maven包装器来执行package goal：

```
$ mvnw package
```

构建成功的话，WAR文件将会出现在target目录下。如果使用Gradle来构建项目，那么可以使用Gradle包装器来执行build任务：

```
$ gradlew build
```

构建完成之后，我们可以在build/libs目录下找到WAR文件。剩下的事情就是部署应用了。不同应用服务器的部署过程会有所差异，所以请参考应用服务器部署过程的相关文档。

比较有意思的事情是，虽然我们构建了适用于Servlet 3.0（或更高版本）部署的WAR文件，但是这个WAR文件依然可以像可执行JAR文件那样在命令行中执行：

```
$ java -jar target/ingredient-service-0.0.19-SNAPSHOT.war
```

实际上，使用一个部署制件，我们同时实现了两种部署方案。

将微服务放到应用服务器中？

按照我们的初衷，更大的Taco Cloud应用是由多个微服务应用组成的，而配料服务只是其中之一。但是，在这里，我们所讨论的是将配料服务部署为一个单独的应用，并将其放到应用服务器中。这样做是合理的吗？

微服务通常和其他的应用一样，应该可以独立部署。尽管离开了Taco Cloud应用的上下文之后，配料服务也没有太大的用处，但是没有理由不能将其部署到Tomcat或其他应用服务器中。不过，我们需要注意的是，不要期望它能够具有像部署到云中那样的可扩展性。

虽然WAR文件作为Java部署的主流方案已经有20多年的历史了，但是它们确实是为将应用程序部署到传统Java应用服务器而设计的。按照我们所选择的平台，现代云部署方案并不需要WAR文件，有些甚至都不支持这种格式。随着我们进入云部署的新时代，JAR文件可能是更好的选择。

19.3　推送JAR文件到Cloud Foundry上

服务器的硬件购买和维护成本可能代价高昂。当出现高负载时，恰当地对服务器进行扩展是非常困难的，对有些组织来说，这样做甚至是不允许的。如今，相对于在自己的数据中心运行应用，将应用部署到云中是一种人们广泛关注并且能够节省成本的方案。

我们有多种可选的云方案，但是人们目前最关注的是平台即服务（Platform as a Service，PaaS）。PaaS提供了现成的应用部署平台，其中包含多种可以绑定到应用上的附加服务（比如数据库和消息代理）。除此之外，如果你的应用需要额外的处理能力，那么云平台很容易在运行时对应用进行扩展（或收缩），这是通过添加和移除实例实现的。

Cloud Foundry是一个开源的PaaS平台，起源于Pivotal（Spring框架

和Spring平台中的其他库也都是由这家公司赞助的）。Cloud Foundry最令人关注的一点在于它提供了开源和基于商业的发行版，让我们可以选择如何以及在哪里使用Cloud Foundry。它甚至可以运行在防火墙之内的公司数据中心里面，提供私有云方案。

虽然Cloud Foundry很乐意接受WAR文件，但是对于Cloud Foundry的需要来说，WAR文件格式过于重量级了。更简单的可执行JAR文件更适合部署到Cloud Foundry中。

为了演示如何构建和部署可执行JAR文件到Cloud Foundry，我们将会构建配料服务应用并将其部署到Pivotal Web Services（PWS）上。如果想要使用PWS，我们就需要注册一个账户。PWS提供87美元的免费试用功能，在试用期间甚至不需要提供任何信用卡信息。

注册完PWS之后，我们需要从PWS下载并安装cf命令行工具。我们将会使用cf工具将应用推送至Cloud Foundry。首先，我们需要使用它来登录PWS账号：

```
$ cf login -a https://api.run.pivotal.io
API endpoint: https://api.run.pivotal.io

Email> {your email}

Password> {your password}

Authenticating...
OK
```

非常好！现在，我们已经准备好将配料服务部署到云中了。实际上，这个项目本身现在就可以部署到Cloud Foundry中，我们所需要做的

就是构建并将其推送至云端。

要使用Maven构建项目的话，我们可以使用Maven包装器执行package goal（将会在target目录下得到形成的JAR文件）：

```
$ mvnw package
```

如果使用Gradle，那么我们可以使用Gradle包装器运行build任务（将会在build/libs目录下得到形成的JAR文件）：

```
$ gradlew build
```

现在，剩下的事情就是使用cf命令将JAR文件推送至Cloud Foundry：

```
$ cf push ingredient-service -p target/ingredient-service-0.0.19-SNAPSHOT.
jar
```

cf push命令的第一个参数指定了在Cloud Foundry中该应用的名称。除了其他功能之外，这个名称还会用作应用托管的子域。因此，非常重要的一点在于，我们为应用设置的名称必须是唯一的，避免与Cloud Foundry已部署的应用（包括其他Cloud Foundry用户所部署的应用）冲突。

想一个唯一名称可能会比较麻烦，cf push命令提供了--random-route选项，它会为我们随机生成一个子域。如下的命令展现了如何将配料服务应用推送至一个随机生成的路由：

```
$ cf push ingredient-service \
```

```
    -p target/ingredient-service-0.0.19-SNAPSHOT.jar \
    --random-route
```

在使用--random-route的时候，依然需要应用名称，但是会在该名称上拼接两个随机选择的单词以生成子域。

假设所有的过程都很顺利，应用应该已经部署就位并且可以处理请求了，子域是ingredient-service，那么我们可以通过浏览器访问http://ingredient-service.cfapps.io/ingredients来查看实际效果。在响应中，我们会看到一个可用配料的列表。

在我编写该应用的时候，它会使用嵌入式的Mongo数据库（这样做只是为了测试）来存放配料数据。在生产环境中，我们可能想要使用真正的数据库。在我编写本书的时候，PWS提供了一个完全托管的MongoDB服务，名为mlab。我们可以使用cf marketplace命令查看该服务（以及其他所有可用的服务）。要创建mlab实例，我们可以使用cf create-service命令：

```
$ cf create-service mlab sandbox ingredientdb
```

该命令会按照沙箱（sandbox）服务计划创建一个mlab服务，名为ingredientdb。服务创建完成之后，我们可以使用cf bind-service命令将其绑定到应用上。例如，要将ingredientdb服务绑定至配料服务应用，我们可以使用如下的命令：

```
$ cf bind-service ingredient-service ingredientdb
```

将服务绑定至应用只是为应用提供了如何连接至服务的详情，这是

通过名为VCAP_SERVICES的环境变量实现的。它并没有改变应用使用服务的方式。服务绑定完成之后，我们需要重新部署（re-stage）应用才能使绑定生效：

```
$ cf restage ingredient-service
```

cf restage命令会强制Cloud Foundry重新部署应用并重新计算VCAP_SERVICES的值。在这个过程中，它会发现应用绑定了一个MongoDB服务，就会使用该服务作为应用的后端数据库。

PWS提供了很多可用的服务，我们可以直接将它们绑定到应用中，包括MySQL数据库、PostgreSQL数据库，甚至现成的Eureka和Config Server服务。建议你阅读一下PWS的Marketplace页面，以了解PWS都提供了哪些服务。关于PWS如何使用，可以参考其官网。

对于Spring Boot应用的部署来讲，Cloud Foundry是一个非常棒的PaaS方案。鉴于它与Spring项目之间的关系，所以会在这两者之间提供一些协同的功能。在云中部署应用程序的另一种常见方法是将应用程序打包到Docker容器中，然后发布到云中，在将应用程序推进到AWS这样的基础设施即服务（Infrastructure-as-a-Service, IAAS）平台时更是如此。下面让我们看看如何创建一个携带Spring Boot应用程序的Docker容器。

19.4　在Docker容器中运行Spring Boot

在分发云中部署的各种应用时，Docker已经成为事实标准。很多云

环境都接受以Docker容器的形式部署应用，包括AWS、Microsoft Azure、Google Cloud Platform和Pivotal Web Services（简单举例）。

容器化应用程序（比如使用Docker创建的应用程序）的概念借鉴了现实世界中的联运集装箱。在运输过程中，不管里面的东西是什么，所有的联运集装箱都有一个标准的尺寸和格式。正因为如此，联运集装箱才能够很容易地堆放在船上、火车上或卡车上。按照类似的方式，容器化的应用程序遵循通用的容器格式，可以在任何地方部署和运行，而不必关心里面的应用是什么。

尽管创建Docker镜像并不困难，但是Spotify提供了一个Maven插件，借助它我们可以轻而易举地将Spring Boot的构建结果创建为Docker容器。要使用该Docker插件，需要将其添加到Spring Boot项目pom.xml文件的<build>/<plugins>代码块下：

```
<build>
  <plugins>
  ...
    <plugin>
      <groupId>com.spotify</groupId>
      <artifactId>dockerfile-maven-plugin</artifactId>
      <version>1.4.3</version>
      <configuration>
        <repository>
          ${docker.image.prefix}/${project.artifactId}
        </repository>
        <buildArgs>
          <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
        </buildArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

在<configuration>代码块下，我们设置了一些属性，用于指导如何创建Docker镜像。<repository>描述了在Docker仓库中该Docker镜像的名称。按照这里的设置，其名称是Maven项目的artifact ID，加上Maven属性docker.image.prefix的值作为前缀。项目的artifact ID是Maven已知的，而前缀属性则需要我们进行设置：

```
<properties>
  ...
  <docker.image.prefix>tacocloud</docker.image.prefix>
</properties>
```

以Taco Cloud的配料服务来讲，所形成的Docker镜像在Docker仓库的名称为tacocloud/ingredient-service。

在<buildArgs>元素下面，我们声明镜像要包含Maven构建所生成的JAR文件，在这里使用Maven属性project.build.finalName来确定target目录下JAR文件的名称。

除了提供给Maven构建文件的信息之外，Docker镜像的所有定义都位于一个名叫Dockerfile的文件中。这个文件指明了新镜像要基于哪个基础镜像、要设置的环境变量、要mount的卷以及最重要的入口点（entry point）。入口点也就是基于该镜像的容器在启动时要执行的命令。对于大多数Spring Boot应用来讲，如下的Dockerfile就是一个很好的起点：

```
FROM openjdk:8-jdk-alpine
ENV SPRING_PROFILES_ACTIVE docker
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
```

```
ENTRYPOINT ["java",\
            "-Djava.security.egd=file:/dev/./urandom",\
            "-jar",\
            "/app.jar"]
```

我们将Docker文件逐行拆分，将会看到它包含如下内容。

- FROM指令声明了新镜像要基于哪个基础镜像。新的镜像会扩展基础镜像。在本例中，基础镜像为openjdk:8-jdk- alpine，这是一个基于OpenJDK 8 的容器镜像。
- ENV指令设置了环境变量。我们可以基于激活状态的profile重写一些Spring Boot应用的配置属性，所以在本镜像中，我们将SPRING_PROFILES_ACTIVE环境变量设置为docker，从而确保Spring Boot应用启动时docker是处于激活状态的profile。
- VOLUME指令在容器中创建了一个mount点。在本例中，它在“/tmp”创建了一个mount点，所以需要的话可以将数据写入“/tmp”目录下。
- ARG指令声明了一个要在构建期传入的参数。在本例中，它声明了名为JAR_FILE的参数，与Maven插件<buildArgs>代码块中的参数是相同的。
- COPY指令会将给定路径下的某个文件复制到另外一个路径下。在本例中，它会将Maven插件中声明的JAR文件复制为容器中名为app.jar的文件。
- ENTRYPOINT描述了容器启动的时候要执行什么操作。它以数组的形式指定了要执行的命令行。在本例中，它使用java命令来运行可执行的app.jar。

我们着重介绍一下ENV指令。在任何包含Spring Boot应用程序的容器镜像中设置SPRING_PROFILES_ACTIVE环境变量通常都是一个好办法。这样的话，我们可以配置一些仅在Docker下运行应用时有效的bean

和配置属性。

对于配料服务，我们需要将应用程序链接到运行在单独容器中的Mongo数据库。默认情况下，Spring Data会尝试连接localhost上监听端口27017的Mongo数据库。但是，这种做法只有在本地运行的时候才有效，并不适合容器。因此，我们需要配置spring.data.mongodb.host属性，告诉Spring Data要访问哪个主机上的Mongo。

虽然我们可能还不知道Mongo数据库运行在何处，但是我们可以通过application.yml文件配置Docker特定的配置，让它在docker profile处于激活状态时配置Spring Data连接名为mongo的主机上的Mongo：

```
---
spring:
  profiles: docker

  data:
    mongodb:
      host: mongo
```

随后，当我们启动Docker容器的时候，会将mongo主机映射到一个在不同容器中运行的Mongo数据库上。现在，我们先来构建容器镜像。借助Maven包装器，执行package和dockerfile:build goal来构建JAR文件，然后构建Docker镜像：

```
$ mvnw package dockerfile:build
```

此时，我们可以通过docker images来校验本地镜像仓库中的镜像（为了可读性和适应本书的宽度，这里将CREATED和SIZE列删减掉了）：

```
$ docker images
REPOSITORY                      TAG        IMAGE ID
tacocloud/ingredient-service    latest     7e8ed20e768e
```

在启动容器之前，我们需要启动Mongo数据库的容器。如下的命令显示了运行一个名为tacocloud-mongo的新容器，其中包含Mongo 3.7.9数据库：

```
$ docker run --name tacocloud-mongo -d mongo:3.7.9-xenial
```

现在，我们终于可以运行配料服务容器了，并链接它到刚刚启动的Mongo容器上：

```
$ docker run -p 8080:8081 \
             --link tacocloud-mongo:mongo \
             tacocloud/ingredient-service
```

这里的docker run命令有多个值得介绍的重要组件。

- 因为我们配置了容器中的Spring Boot应用运行在8081端口上，所以-p参数可以将内部端口映射到主机的8080端口上。
- --link参数能够将我们的容器链接到名为tacocloud-mongo的容器上，并为其分配mongo主机名，这样Spring Data就可以使用该主机名连接它了。
- 最后，我们指定了容器要运行的镜像名称（也就是tacocloud/ingredient-service）。

现在，Docker镜像构建完成并且已经证明可以作为本地容器运行。我们可以更进一步，将镜像推送至Dockerhub或其他Docker镜像仓库。如果你有Dockerhub账号并且已经登录，那么可以使用如下的Maven命

令推送镜像：

```
$ mvnw dockerfile:push
```

这样的话，我们可以将镜像部署到几乎所有支持Docker容器的环境中，包括AWS、Microsoft Azure和Google Cloud Platform。你可以选择任意的环境并按照平台相关的指令部署Docker镜像。

19.5 以终为始

在过去的几百页中，我们从一个简单的起点开始（更具体来讲，也就是start.spring.io），最终将应用部署到了云中。我希望在这几百页中，你所能获取到的乐趣与我在编写本书的过程中是一样的。

虽然本书必须要结束了，但是你的Spring征程才刚刚开始。利用本书所学的知识，用Spring构建令人赞叹的应用吧！我迫不及待地想知道你们的成就。

19.6 小结

- Spring应用可以部署到多种不同的环境中，包括传统的应用服务器、像Cloud Foundry这样的平台即服务环境，或者Docker容器。
- 在构建WAR文件的时候，我们应当包含一个SpringBootServletInitializr的子类，确保Spring的DispatcherServlet恰当地进行了配置。
- 构建可运行的JAR文件允许将Spring Boot应用部署到多个云平台上，而且能够避免WAR文件带来的开销。

- 借助Spotify的Maven Dockerfile插件，能够非常容易地容器化Spring应用。它会在Docker容器中包装一个可执行的JAR文件，容器可以部署到任何支持Docker的环境中，其中包括像Amazon Web Services、Microsoft Azure、Google Cloud Platform、Pivotal Web Services （PWS）、Pivotal Container Service （PKS）这样的云供应商。

附录　初始化Spring应用

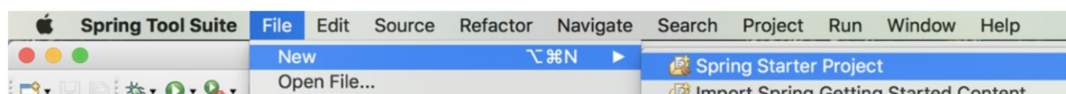
有很多种方式都可以初始化Spring项目，至于选择哪一种完全取决于个人喜好。其中，很多方案是由我们喜欢哪款IDE决定的。

我们在这里所讨论的可选方案除了一种之外，其他的都是基于Spring Initializr的，Spring Initializr是一个能够为我们生成Spring Boot项目的REST API。各种IDE只不过是REST API的客户端。另外，还有几种方式可以在IDE之外使用Spring Initializr API。

在本附录中，我们将会快速看一下所有的可选方案。

A.1　使用Spring Tool Suite初始化项目

要使用Spring Tool Suite来初始化新的Spring项目，我们需要在File > New菜单中选择Spring Starter Project菜单项，如图A.1所示。



图A.1　在Spring Tool Suite中初始化一个新项目

> 注意：这是一个使用Spring Tool Suite初始化Spring项目的简单描述。更详细的阐述，可以参考1.2.1小节。

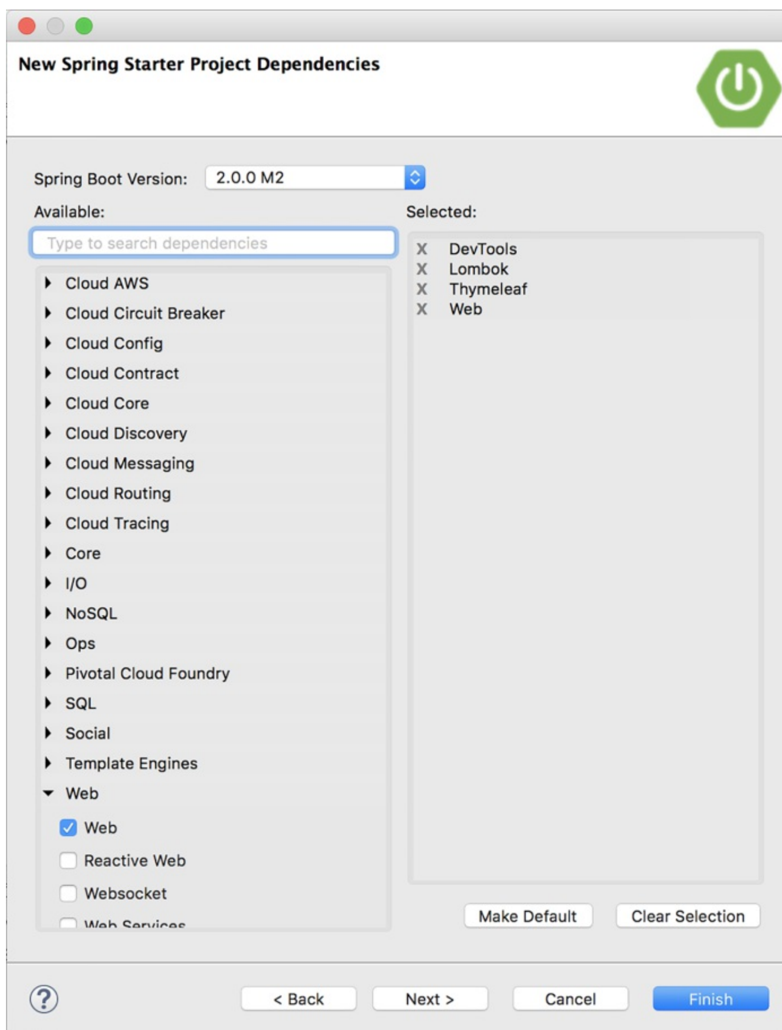
接下来，我们将会看到项目创建对话框的第一页（见图A.2）。在这个页面中，我们将会定义项目的基本信息，比如项目名称、坐标（group ID和artifact ID）、版本和基础包名。我们也可以确定项目使用Maven还是Gradle来构建，还可以声明构建生成JAR文件还是WAR文件，以及使用哪个版本的Java，甚至还能使用其他的JVM语言，比如Groovy或Kotlin。

图A.2　定义基本的项目信息

这个页面的第一个输入域要求我们指定Spring Initializr服务的位

置。如果你运行或使用自定义的Initializr实例，那么可以在这里指定Initializr服务的基础URL；否则，使用默认的http://start.spring.io就可以了。

在定义完项目的基本信息之后，点击Next按钮，会看到项目的依赖页（见图A.3）。



图A.3　指定项目的依赖

在项目依赖页中，我们可以指定项目需要的所有依赖。其中，有些

依赖是Spring Boot Starter依赖，有些依赖是Spring项目常用的依赖。

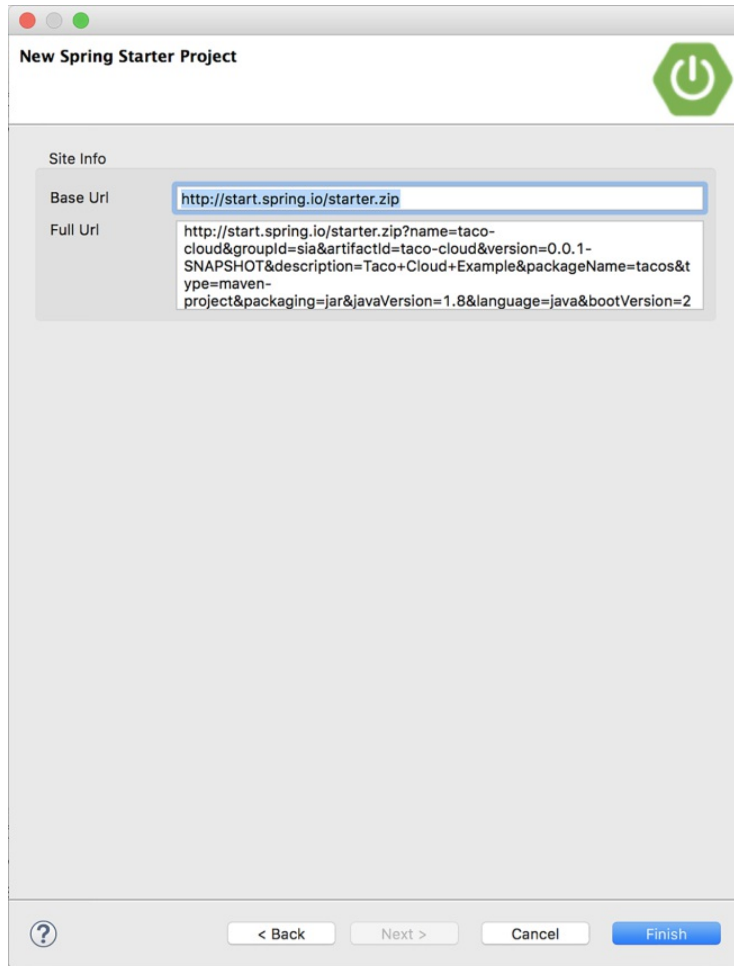
可用的依赖都列在左侧，以分组的形式进行组织，可以展开和收缩。如果在查找依赖时遇到麻烦，还可以对依赖进行搜索，以便于缩小可选范围。

要将某个依赖添加到所生成的项目中，我们只需要选中依赖名称前面的复选框即可。已经选中的依赖会显示在右侧Selected标题下面。如果想要移除依赖，可以点击已选中依赖前面的×，也可以点击Clear Selection按钮清除所有已选的依赖。

为了增加便利性，如果你发现在项目中特定的一组依赖始终（或者经常）会用到，那么你可以在选择完这些依赖后点击Make Default按钮，这样在下一次创建项目的时候它们会预先选中。

在选择完之后，点击Finish按钮就可以生成项目并添加到工作空间中了。

如果你想要使用http://start.spring.io之外的其他Initializr，可以点击Next按钮来设置Initializr的基础URL，如图A.4所示。

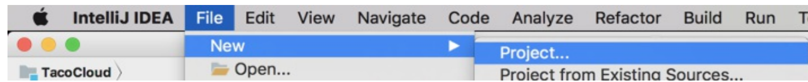


图A.4　指定Initializr的基础URL

Base Url输入域指定了Initializr API监听的URL。在这个页面中，这是唯一可以修改的输入域。Full Url输入域展现了通过Initializr请求新项目的完整URL地址。

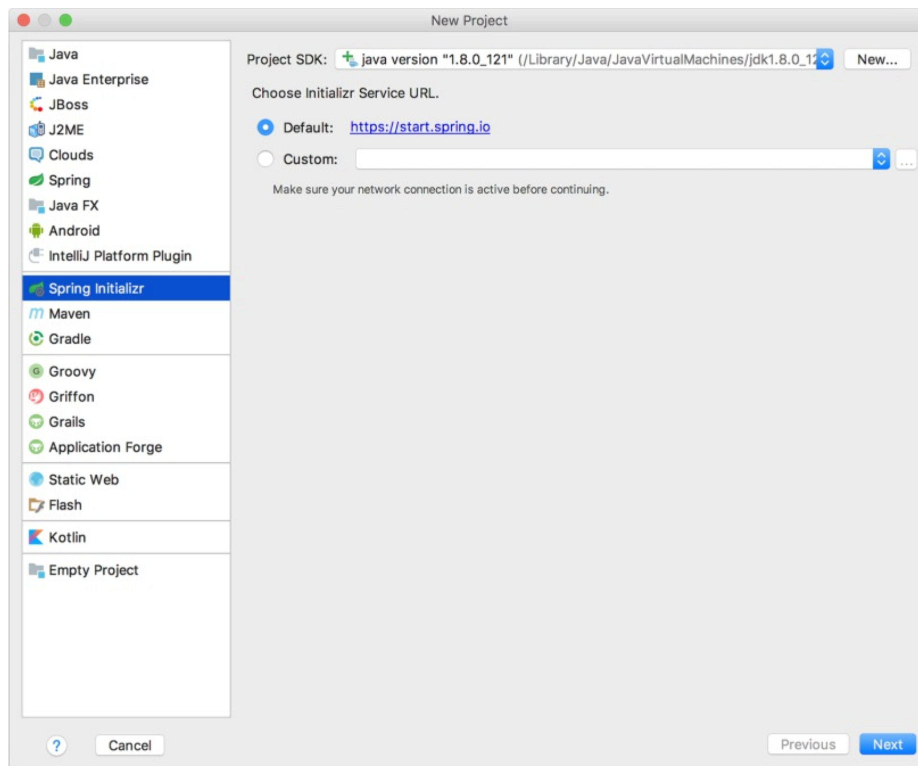
A.2　使用IntelliJ IDEA初始化项目

如果要使用IntelliJ IDEA初始化Spring项目，就在File > New菜单下选择Project菜单项，如图A.5所示。



图A.5　在IntelliJ IDEA中初始化一个新的Spring项目

此时，将会打开新Spring Initializr项目向导的第一页（参见图A.6）。在本页中，通常我们会直接点击Next按钮进入下一页。如果你想要使用与https://start.spring.io不同的Spring Initializr，就可以选中Custom单选框，并输入想要使用的Spring Initializr的基础URL。



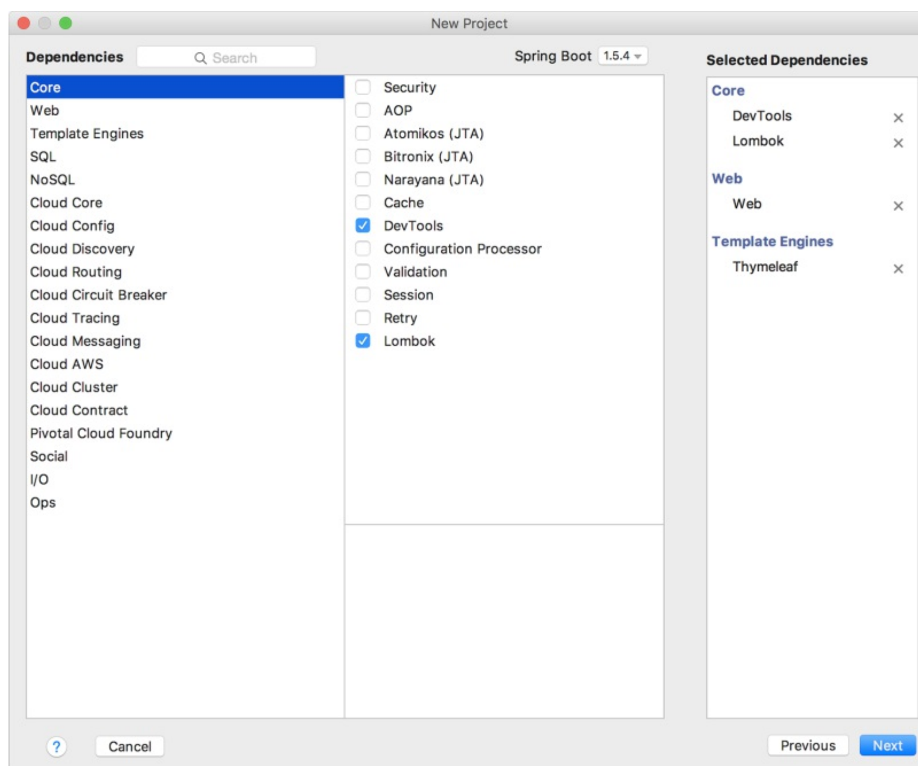
图A.6　选择Spring Initializr的位置

点击Next按钮之后，我们将会看到一个输入项目基本信息的页面，如图A.7所示。你会发现，这个页面上有很多输入域与Maven pom.xml中的信息是一致的。实际上，在Type输入域中选择Maven Project，就能发现这些输入域的用途所在。如果你更喜欢Gradle，也可以选择Gradle

Project。

图A.7　在IntelliJ IDEA中指明必要的项目信息

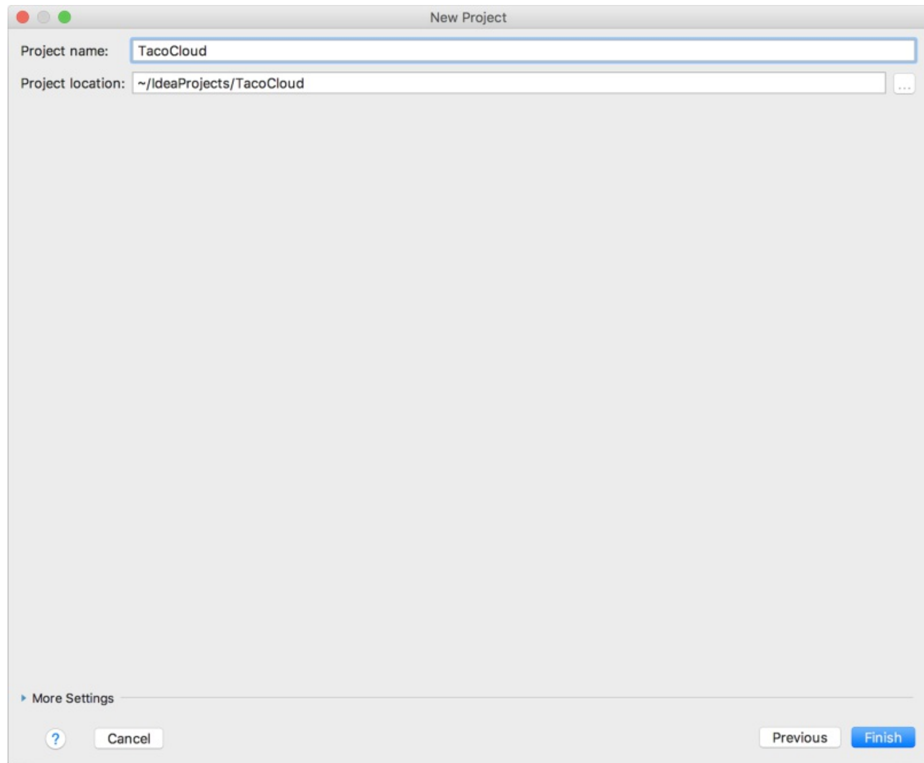
在填写完必要的项目信息后，点击Next按钮将会展现项目依赖页（见图A.8）。



图A.8　选择项目依赖

在最左侧，依赖是按照分类来组织的。选中某个分类时，这个分类对应的可选项会显示在中间区域。已经选中的依赖将会（按照分类）列在最右侧。

在选择完依赖之后，点击Next按钮，我们将会看到项目向导的最后一页，如图A.9所示。在这个页面中，我们要输入项目的名称并指定项目要放到磁盘的什么位置。

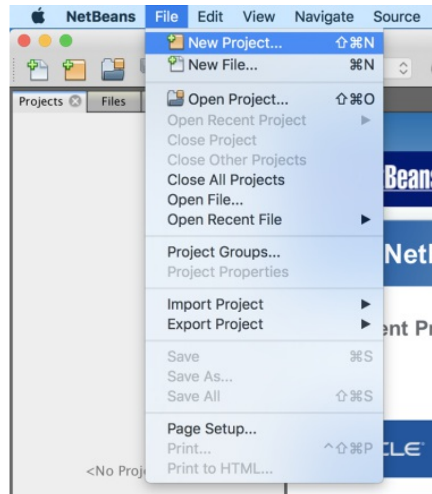


图A.9　设置项目的名称和位置

点击Finish按钮，项目将会创建并加载到IntelliJ IDEA的工作空间中。

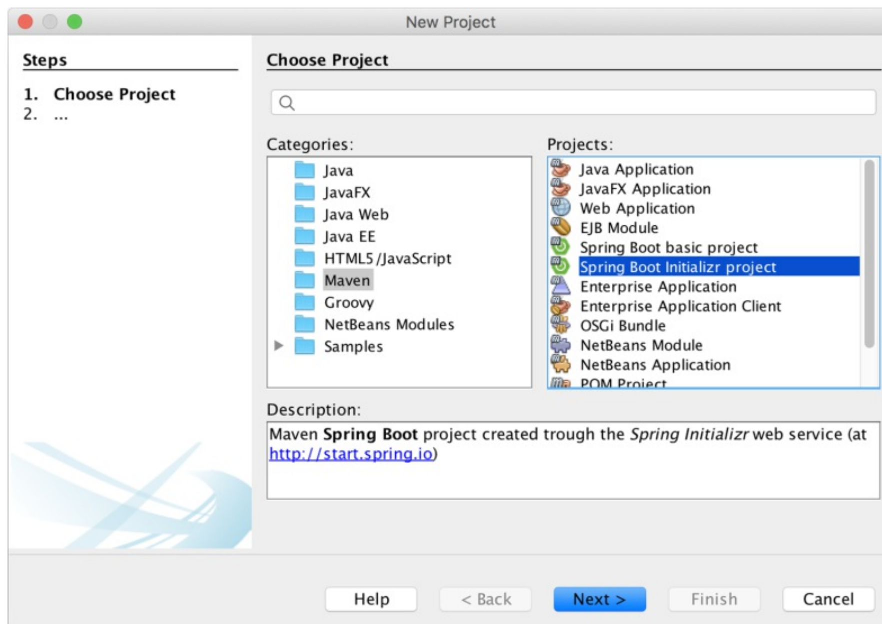
A.3　使用NetBeans初始化项目

要使用NetBeans创建新项目，首先要选择File菜单的New Project菜单项，如图A.10所示。



图A.10　使用NetBeans初始化一个新的Spring项目

此时我们会看到新项目向导的第一页，如图A.11所示。该页面会让我们选择想要创建什么类型的项目。



图A.11　创建新的Spring Boot Initializr项目

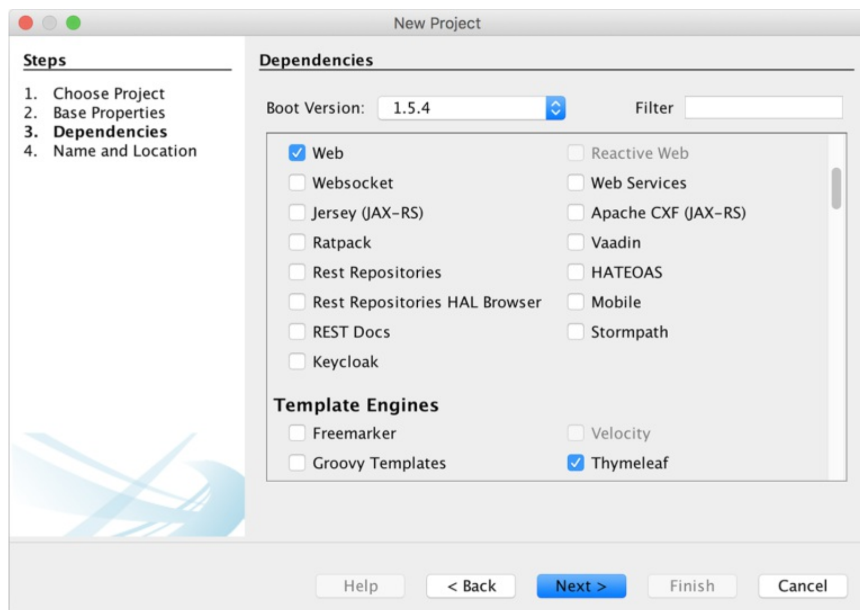
对于Spring Boot项目来说，我们要从左侧的列表中选择Maven，然后在右侧的项目列表中选择Spring Boot Initializr Project，然后点击Next

按钮进入下一页。

新项目向导的第二页（见图A.12）允许我们设置项目的基本信息，比如项目名称、版本以及Maven pom.xml文件中定义项目的其他信息。

图A.12　声明项目的基本信息

在声明完项目的基本信息之后，点击Next按钮进入新项目向导的依赖页，参见图A.13。

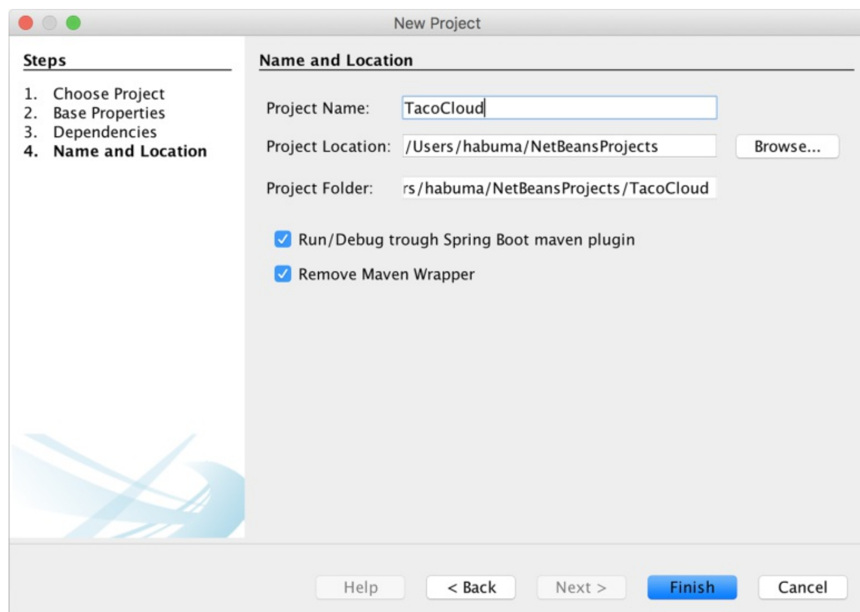


图A.13　选择项目的依赖

依赖会按照分类的形式全部列在同一个列表中。如果在查找特定依赖时遇到麻烦，可以使用顶部的Filter文本框限制列表中可选项的数量。

在这个页面中，还可以指定想要使用哪个Spring Boot版本。它默认会设置为当前Spring Boot的正式版本。

在为项目选择完依赖之后，点击Next按钮会显示新项目向导的最后一页，如图A.14所示。这个页面允许我们声明项目的一些详情信息，包括项目的名称以及文件系统中的位置（Project Folder文本域是只读的，它的值会根据其他两个文本域的值衍生而来）。在这里，还允许我们通过Maven Spring Boot插件运行和调试项目，而不是使用NetBeans。我们还可以让NetBeans移除生成项目中的Maven包装器。



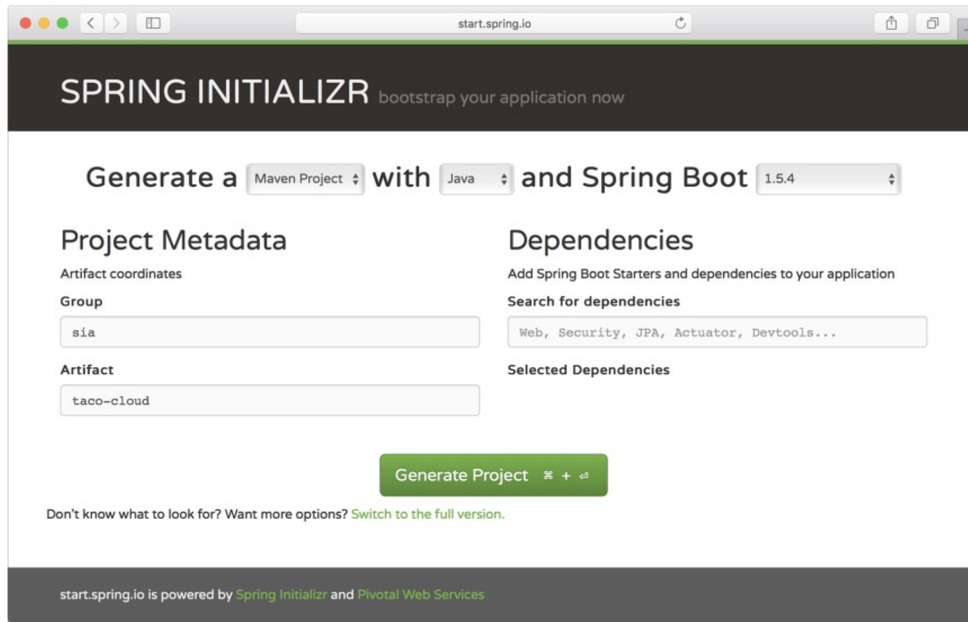
图A.14　指明项目的名称和位置

在设置完项目的所有信息后，点击Finish按钮，项目将会创建并添加到NetBeans的工作空间中。

A.4　在start.spring.io中初始化项目

到目前为止所描述的基于IDE的初始化方案可能会满足你的需求，但是有时候你可能需要完全不同的IDE，或者使用简单的文本编辑器。在这种情况下，你依然可以借助基于Web界面的Initializr来使用Spring Initializr。

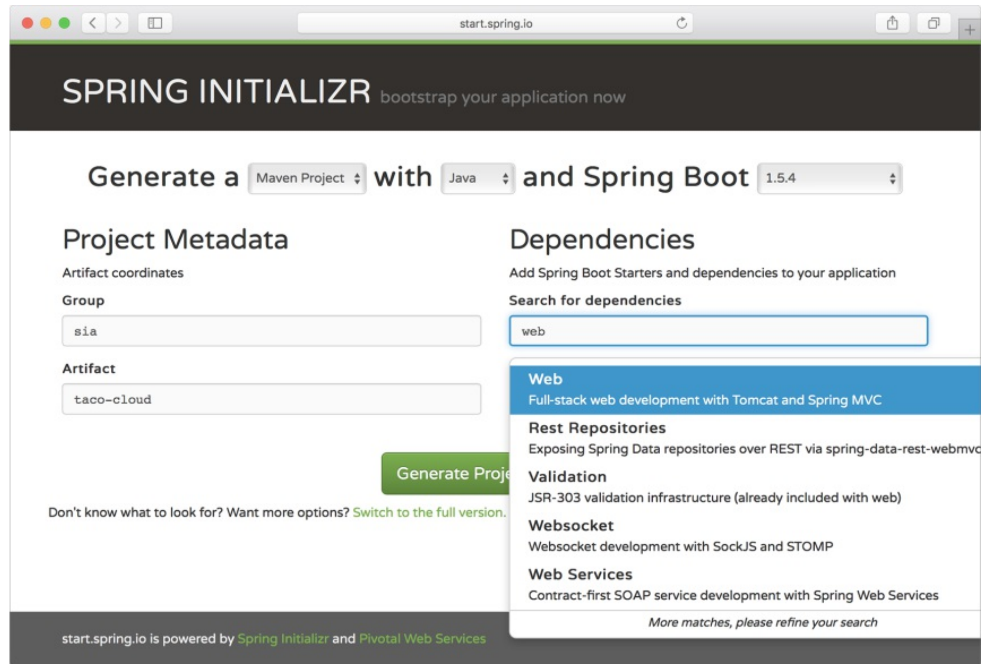
首先，在Web浏览器中访问https://start.spring.io。我们将会看到简单版本的Spring Initializr Web用户界面，如图A.15所示。



图A.15　简单版本的Spring Initializr Web界面

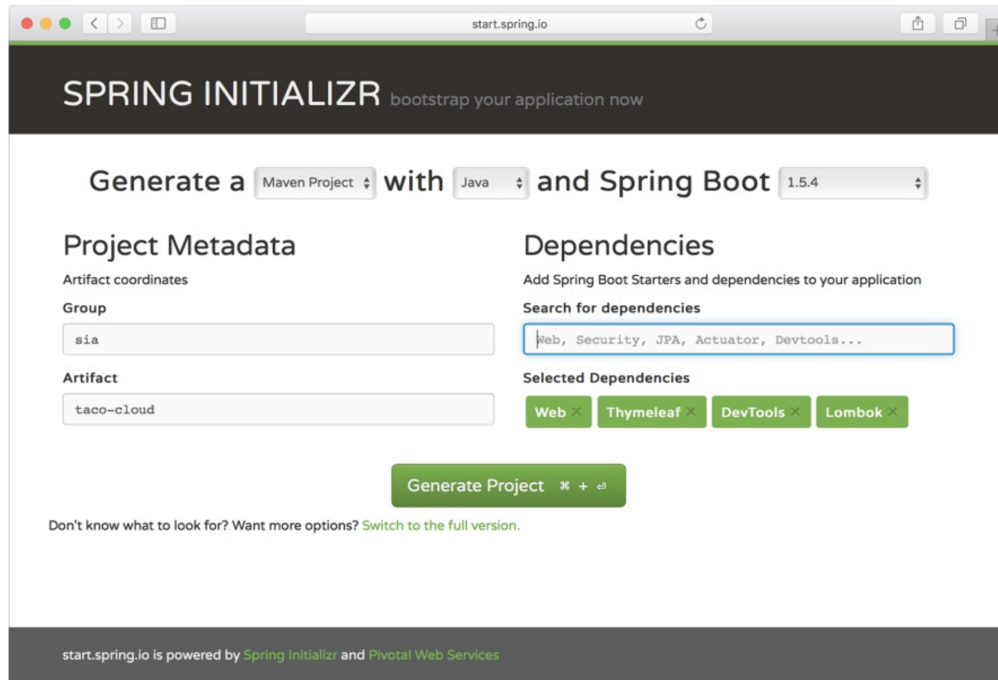
在简单版本的Initializr Web应用中，我们只需要填写一些非常基本的信息，比如使用Maven还是Gradle进行构建、开发项目要使用什么语言、基于什么版本的Spring Boot进行构建以及项目的group和artifact ID。

我们还可以通过在Search for Dependencies文本框中输入搜索条件来指明依赖。例如，如图A.16所示，我们可以输入“web”来搜索带有“web”关键字的依赖。



图A.16　搜索依赖

当看到自己想要的依赖时，在键盘上按Return键来选中它，它就会添加到选中依赖的列表中。在图A.17中，Selected Dependencies文本下面显示已经选中了Web、Thymeleaf、DevTools和Lombok依赖。



图A.17　选择依赖

如果不想要某个已选中的依赖，只需要点击依赖条目右侧的×就可以将其移除。

完成之后，我们可以点击Generate Project按钮（也可以使用按钮上所显示的快捷键，不同操作系统下会有所差异），让Initializr初始化项目并下载为zip文件。然后，我们就可以解压该文件并导入你所选择的IDE或文本编辑器了。

如果你希望更细粒度地控制项目创建的过程，可以点击Generate Project下的Switch to the full version链接，这样用户界面会展现更多的输入域和所有可用依赖的复选框列表。图A.18显示了Web界面完整版本的一部分。

图A.18　完整版本的Initializr用户界面（部分）

完整版本中的大多数输入域都衍生自Group和Artifact输入域，或者在简单版本中已经有了默认值。完整版本能够让我们重写这些衍生值/默认值。

图A.18只展现了完整版本中可用依赖复选框的一部分，所以你可能需要向下滑动很久才能找到想要的依赖。不过，在完整版本的用户界面中，搜索框依然是可用的。

A.5 使用命令行初始化项目

Spring Initializr的IDE和基于浏览器的用户界面可能是初始化项目的常见方式。它们都是Initializr应用程序提供的REST服务的客户端。在某些特殊情况下（例如，在脚本化场景中），我们可能会发现直接从命令行使用Initializr服务也很有用。

我们有两种消费该API的方式：

- 使用curl命令（或者类似的命令行REST客户端）；
- 使用Spring Boot的命令行接口（又称为Spring Boot CLI）。

下面我们看一下这两种方案，先从curl命令开始。

A.5.1 curl和Initializr API

使用curl初始化Spring项目的最简单方式是按照如下格式消费该API：

```
% curl https://start.spring.io/starter.zip -o demo.zip
```

在本例中，我们请求了Initializr的“/starter.zip”端点，它将会生成一个Spring项目并下载为zip文件。生成的项目是使用Maven构建的，并且除了Spring Boot starter依赖外并没有其他的依赖，pom.xml文件中的所有项目信息都是默认值。

如果不进行特殊指定，文件名将会是starter.zip。在本例中，-o选项

将下载的文件命名为demo.zip。

对外公开的Spring Initializr服务器托管在https://start.spring.io上，如果你想要使用自定义Initializr，那么需要对应地修改URL。

除了默认值之外，我们可能还想要指定一些详情信息和依赖。表A.1列出了消费Spring Initializr REST服务所有可用的参数（及其默认值）。

表**A.1　Initializr API**支持的请求参数

参数	描述	默认值
groupId	项目的group ID，用于Maven仓库对各种制件的组织	com.example
artifactId	项目的artifact ID，将会显示在Maven仓库中	demo
version	项目版本	0.0.1-SNAPSHOT
name	项目名称，同时也会用来确定应用主类的名称（类名会添加Application后缀）	demo
description	项目的描述	Demo project for Spring Boot
packageName	项目的基础包名	com.example.demo

dependencies	项目构建文件所包含的依赖	基础的Spring Boot starter
type	所生产项目的类型，maven-project或gradle-project	maven-project
javaVersion	基于哪个Java版本进行构建	1.8
bootVersion	基于哪个Spring Boot版本进行构建	当前GA版本的Spring Boot
language	要使用的编程语言，可以是java、groovy或kotlin	java
packaging	项目应该如何打包，可以是jar或war	jar
applicationName	应用的名称	name参数的值
baseDir	在生成的归档文件中基础路径的名称	根路径

我们可以通过发送请求至基础Initializr URL获取参数列表和可用依赖项的列表：

```
% curl https://start.spring.io
```

在这些参数中，我们可能会发现dependencies是最常用的。例如，

我们想要创建一个使用Spring的简单Web项目。如下使用curl的命令行将会生成一个包含web starter依赖的项目zip包：

```
% curl https://start.spring.io/starter.zip \
      -d dependencies=web \
      -o demo.zip
```

假设有一个更复杂的样例：我们想要开发一个使用Spring Data JPA进行数据持久化的Web应用程序，使用Gradle构建，位于zip文件中名为my-dir的目录下，并且我们不仅想要下载zip文件，还希望在下载时将项目解压到文件系统中。在这种情况下，下面的命令可以解决该问题:

```
% curl https://start.spring.io/starter.tgz \
       -d dependencies=web,data-jpa \
       -d type=gradle-project
       -d baseDir=my-dir | tar -xzvf -
```

在这里，下载的zip文件将会以管道的方式传递给tar命令进行解压。

A.5.2 Spring Boot命令行接口

Spring Boot CLI是另一种初始化Spring应用的方案。我们能够以多种方式安装Spring Boot CLI，最简单的（也是我最喜欢的）方式可能是使用SDKMAN：

```
% sdk install springboot
```

Spring Boot CLI安装完成之后，我们就可以使用它来生成项目了。

使用方式与curl非常类似，这里我们要使用的命令是spring init。实际上，使用Spring Boot CLI生成项目的最简单方式为：

```
% spring init
```

这样会生成一个Spring Boot项目的骨架，并且会下载成名为demo.zip的zip文件。

有时我们可能想要指明一些详情信息和依赖，表A.2列出了spring init命令所有可用的参数。

表**A.2　spring init**命令支持的所有请求参数

参数	描述	默认值
group-id	项目的group ID，用于Maven仓库对各种制件的组织	com.example
artifact-id	项目的artifact ID，将会显示在Maven仓库中	demo
version	项目版本	0.0.1-SNAPSHOT
name	项目名称，同时也会用来确定应用主类的名称（类名会添加Application后缀）	demo
description	项目的描述	Demo project for Spring Boot

package-name	项目的基础包名	com.example.demo
dependencies	项目构建文件所包含的依赖	基础的Spring Boot starter
type	所生产项目的类型，maven-project或gradle-project	maven-project
java-version	基于哪个Java版本进行构建	1.8
boot-version	基于哪个Spring Boot版本进行构建	当前GA版本的Spring Boot
language	要使用的编程语言，可以是java、groovy或kotlin	java
packaging	项目应该如何打包，可以是jar或war	jar

通过使用--list参数，我们可以列出参数的列表以及可用依赖：

```
% spring init --list
```

假设我们希望创建一个基于Java 1.7的Web应用，那么如下使用--dependencies和--java命令即可：

```
% spring init --dependencies=web --java-version=1.7
```

假设我们想要创建一个使用Spring Data JPA进行数据持久化的Web

应用程序，并且还希望使用Gradle构建它，而不是使用Maven，那么我们可以使用如下的命令：

```
% spring init --dependencies=web,jpa --type=gradle-project
```

你可能已经发现，spring init的很多参数与curl方案的参数相同或相似。也就是说，spring init并没有支持curl方案中的所有参数（比如baseDir），而且参数是中划线分割的而不是采用驼峰命名（例如package-name与packageName）。

A.6　使用元框架创建Spring应用

另外值得一提的是，有一些基于Spring和Spring Boot构建的框架：

- Grails。
- JHipster。

这些元框架（meta-framework）在更高层级上提供了快速开发Spring应用的能力，同时依然能够使用Spring和Spring Boot所提供的所有功能。

这些元框架都有自己独特的开发模型，实际上，它们本身就是框架，因此在本附录中简单地将它们作为项目初始化机制有点不公平。事实上，每个元框架都值得写一本书。

我们不会深入研究如何使用这些元框架来初始化Spring项目，在这里将它们列出来只是让读者知道还有初始化和开发Spring应用的其他方

式。

A.7　构建和运行项目

不管你采用什么方式初始化项目，都可以在命令行中使用java -jar命令来运行应用：

```
% java -jar demo.jar
```

即使不采用JAR文件而使用WAR文件来进行分发，这样做也依然是可行的：

```
% java -jar demo.war
```

我们还可以使用Spring Boot Maven或Gradle插件来运行应用。比如，项目使用Maven构建的话，可以这样运行：

```
% mvn spring-boot:run
```

使用Gradle进行构建的话，可以按照如下方式运行项目：

```
% gradle bootRun
```

不管是采用Maven还是Gradle，构建工具都会构建（还没有构建的话）并运行项目。